(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

A DESIGN OF FLOATING POINT FFT USING GENESIL SILICON COMPILER

by

Lu, Chung-Kuei

June 1991

Thesis Advisor: Chyan Yang

Approved for public release; distribution is unlimited.

Unclassified

Security Classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification: Unclassified	1b Restrictive Markings
2a Security Classification Authority	3 Distribution Availability of Report: Approved for public release; distribution is unlimited.
2b Declassification/Downgrading Schedule	
4 Performing Organization Report Number(s)	5 Monitoring Organization Report Number(s)
6a Name of Performing Organization: Naval Postgraduate School; 6b Office Symbol (*If Applicable*): EC	7a Name of Monitoring Organization: Naval Postgraduate School
6c Address (*city, state, and ZIP code*): Monterey, CA 93943-5000	7b Address (*city, state, and ZIP code*): Monterey, CA 93943-5000
8a Name of Funding/Sponsoring Organization; 8b Office Symbol (*If Applicable*)	9 Procurement Instrument Identification Number
8c Address (*city, state, and ZIP code*)	10 Source of Funding Numbers: Program Element Number; Project No; Task No; Work Unit Accession No

11 Title (*Include Security Classification*)
A DESIGN OF FLOATING POINT FFT USING GENESIL SILICON COMPILER

12 Personal Author(s)
Lu, Chung-Kuei

13a Type of Report	13b Time Covered	14 Date of Report (*year, month,day*)	15 Page Count
Master's Thesis	From To	June1991	76

16 Supplementary Notation: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes		
Field	Group	Subgroup

18 Subject Terms (*continue on reverse if necessary and identify by block number*)
Fast Fourier Transform; Genesil Silicon Compiler; Floating-point; Sign; Exponent; Fraction; Normalization; Alignment; Overflow; Underflow; Functional Simulation; Timing Analysis; Output Delay; Sign-magnitude; Two's complement

19 Abstract (*continue on reverse if necessary and identify by block number*)

The hardware of floating-point MULTIPLY, ADD, and SUBTRACT units are designed to support the multiplication, addition, and subtraction operation necessary in the Fast Fourier Transform (FFT).

In this thesis, the IEEE floating-point standard is adopted and scaled down to 16 bits, but the exponent is an excess-8 number represented using radix-2. A 16 bit reduced word size floating-point arithematic unit for high speed signal analysis was implemented. The layout verification, functional simulation, and timing analysis of these units have been performed on the Genesil Silicon Compiler (GSC) system that was developed to overcome the shortcomings of the time consuming custom layout methods. The design of this thesis work can be used for further investigation of the high speed, pipelined floating-point arithmetic units.

20 Distribution/Availability of Abstract	21 Abstract Security Classification
[X] unclassified/unlimited [ ] same as report [ ] DTIC users	Unclassified

22a Name of Responsible Individual	22b Telephone (*Include Area code*)	22c Office Symbol
Yang, Chyan	(408) 646-2266	EC/Ya

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted

All other editions are obsolete

security classification of this page

Unclassified

Approved for public release; distribution is unlimited.

A Design of Floating Point FFT Using Genesil Silicon Compiler

by

Lu, Chung-Kuei
Lieutenant Commander, Republic of China Navy
B.S., Chinese Naval Academy, 1981

Submitted in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1991

Author: _____
Lu, Chung-Kuei

Approved by: _____
Chyan Yang, Thesis Advisor

_____
Herschel H. Loomis, Jr., Thesis Co-advisor

_____
Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

The hardware of floating-point MULTIPLY, ADD, and SUBTRACT units are designed to support the multiplication, addition, and subtraction operation necessary in the Fast Fourier Transform (FFT).

In this thesis, the IEEE floating-point standard is adopted and scaled down to 16 bits, but the exponent is an excess-8 number represented using radix-2. A 16 bit reduced word size floating-point arithmetic unit for high speed signal analysis was implemented. The layout verification, functional simulation, and timing analysis of these units have been performed on the Genesil Silicon Compiler (GSC) system that was developed to overcome the shortcomings of the time consuming custom layout methods. The design of this thesis work can be used for further investigation of the high speed, pipelined floating-point arithmetic units.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to extend a heartfelt thank you to the following individuals for their time, patience, and energy helping make this thesis come to fruition:

Prof. Chyan Yang, ECE. Naval Postgraduate School

Prof. Herschel H. Loomis, Jr., ECE. Naval Postgraduate School

David E. Gilbert, USN, Class of 1991, Naval Postgraduate School

Kim Chen, MSEE, Class of 1990, Naval Postgraduate School

And a very special thank you to my wife Lulee, Pi-Hwa for her love and support which allowed me to successfully complete my graduate education.

I. INTRODUCTION

A. BACKGROUND

In this section, the basic concepts of The Fast Fourier Transform and The Genesil Silicon Compilier System are introduced.

1. The Fast Fourier Transform

For some applications (e.g., speech, radar signal,..., etc), analysis in the frequency domain is simpler than that in the time domain. In signal processing and spectral analysis the frequency domain is used. The fast fourier transform (FFT) is used to transform data from the time domain into the frequency domain. Availability of special-purpose hardware in both areas has led to sophisticated signal-processing systems based on the features of the FFT.

The popularity of the FFT is evidenced by the wide areas of application. In addition to conventional radar, communications, sonar, and speech signal-processing applications, current applications of FFT usage include biomedical engineering, imaging, metallurgical analysis, mechanical analysis, geophysical analysis, simulation, music synthesis, and the determination of weight variation in the production of paper from pulp [Ref. 1]. Note that The radar and communication signal processing demands high speed FFT computation.

2. The Genesil Silicon Compiler System

The Genesil Silicon Compiler (GSC) system is design automation software system that allows systems engineers and circuit designers to design

complex Very Large Scale Integration (VLSI) computer chips. GENESIL produces Integrated Circuited (IC) designs from architectural descriptions and allows for their verification.

The GSC is based on an object-oriented hierarchical system running under the UNIX operating system. The objects consists of Blocks, Modules, Chips, and Chip-sets. The GSC allows a designer to easily create complex circuits using devices from the compiler library. This design procedure does not require the expertise and tedium involved with design at the transistor level [Ref. 2.].

The advantage that a silicon-compiler-based process has over a custom IC system design process is that the latter requires a team of experts in the fields of logic implementation, circuit simulation, chip layout, and testing. However, the design process based on the silicon compiler may be accomplished by the individual utilizing a top-down hierarchical design methodology beginning with a partitioned chip set, progressing downward into individual chips and modules, and terminating at the block level. There is far less time required to design an IC using a silicon compiler than for a full custom manual CAD method using graphic layout tools. This is especially attractive for military applications where small numbers of special purpose chips are required and a rapid turnaround time is desired. Thus, one can see that the silicon compiler provides a streamlined method for rapid development of IC products. The disadvantages, however, of the silicon compiler are that the resulting circuit is often slower and the layout is not always efficient in its use of silicon area [Ref. 3].

B. THESIS GOALS AND ORGANIZATION

The main goal of this thesis was to design a floating-point FFT unit using the GENESIL Silicon Compiler. The motivation for this thesis was to explore the feasibility of designing the FFT with floating-point arithmetic units using state-of-the-art VLSI circuit design tools. The investigation was thorough and the basic design was developed. The floating-point number system and FFT structure will be introduced In Chapter II. The design process of floating-point multiplication and addition will be discussed in Chapter III. The results of the simulation of floating-point multiplication, addition, and FFT will be presented in Chapter IV. Chapter V includes the conclusions of this thesis and the recommendations for future investigation.

II. FLOATING-POINT FFT

In this chapter, we will introduce the floating-point number system in section A, the floating-point arithmetic in section B, and the FFT structure in section C. A detailed description of the hardware design will be discussed in Chapter III, and some examples, which include the special cases, will be presented in Chapter IV.

A. FLOATING-POINT NUMBER SYSTEM

There are four binary number systems most commonly considered for use in fixed point arithmetic operations [Ref. 4]:

- sign-magnitude,
- ones' complement,
- two's complement, and
- excess 2^{m-1}

Traditionally, the excess number system is used to represent the exponen of floating-point number. The range of excess representation is exactly the same as that of two's complement numbers. In fact, the representation for the sign bits, the excess and two's complement number systems are always opposite. A 4-bit comparison table of two's complement and excess representation is shown in the Table 2.1.

The sign-magnitude system uses the most significant bit to represent the sign of the number; all other bits represent the magnitude. It is possible to use any of the fixed point representation schemes for the fracton of a floating-point number. Each has its own advantages and disadvantages. In this thesis,

TABLE 2. 1. A 4-BIT COMPARISON TABLE OF TWO'S COMPLEMENT AND EXCESS-8 REPRESENTATION

	Two's Complement value	Excess - 8 value
0000	0	-8
0001	1	-7
0010	2	-6
0011	3	-5
0100	4	-4
0101	5	-3
0110	6	-2
0111	7	-1
1000	-8	0
1001	-7	1
1010	-6	2
1011	-5	3
1100	-4	4
1101	-3	5
1110	-2	6
1111	-1	7

the sign and fraction of floating-point number are described by the sign-magnitude system.

The single-precision arithmetic refers to those operations defined over standard data operands with word length equal to that of one memory word. The word length, 4, 8, 12, 16,..., 32,..., could be used, because of typical memory chip sizes. But the fraction part (including sign bit) of the word should not be

less than 12 bits; in fact, the 12 bits' quantization is often too coarse for signal spectral analysis. However, the goal of this thesis is to design a high speed arithmetic chip, and reducing the word size increases arithmetic speed.

Therefore, we use the minimal word length, 16 bits, to represent the floating-point numbers as follows:

- 1 bit for the sign,
- 4 bits for the exponent, and
- 11 bits for the fraction.

The IEEE floating-point standard is adopted and scaled down to 16 bits, but the exponent is an excess-8 number represented using radix-2. The fraction represents a number less than one, but the *significant* of the floating-point number is one plus the fraction part. In other words, if e is the value of the exponent and f is the value of the fraction, the magnitude being represented is $1.f \times 2^{e-8}$. The fractional part of a floating-point number must not be confused with the significant, which is one plus the fractional part. The leading 1 in the significant $1.f$ does not appear in the representation; that is, the leading bit is implicit. This is often referred to as the hidden one technique. When performing arithmetic on numbers, the fraction part normally needs to be made explicit [Ref. 5]. The representable range of the floating-point numbers is shown in Figure 2.1.

B. FLOATING-POINT ARITHMETIC

Differences between the floating-point arithmetic and the integer arithmetic are as follows: an exponent field must be manipulated, in addition to the fraction field, and the result of a floating-point operation usually has to

maximum value	$(1.111\ldots)_2 \times 2^{+7} = (255.9375)_{10} = V_{max}$
minimum value	$(1.000\ldots)_2 \times 2^{-7} = (.0078125)_{10} = V_{min}$
true zero	$e = 0$ and $f = 0$
zero	$e = 0$ and $f \neq 0$

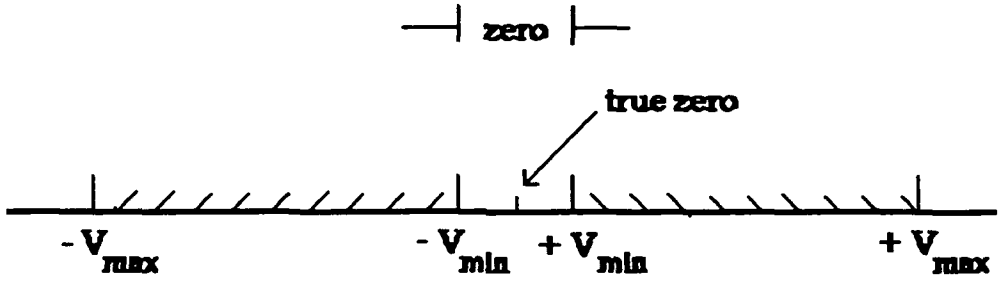


Figure 2.1 Ranges of 16 Bit Numbers in Floating Point Representations

be rounded to be represented by another floating-point number of the same precision.

1. Floating-point Multiplication

Floating-point multiplication is perhaps the simplest floating-point operation in terms of the required hardware. No alignment of the operands are required before initiating the operation, and minimal normalization is required at the end of the operation.

The logic of floating-point multiplication of two numbers is shown in Figure 2.2 [Ref. 6]. The fraction of the result is the product of the multiplier and multiplicand fractions; the exponent of the result is the sum of the exponents of the input operands; the sign bit of the result is simply the

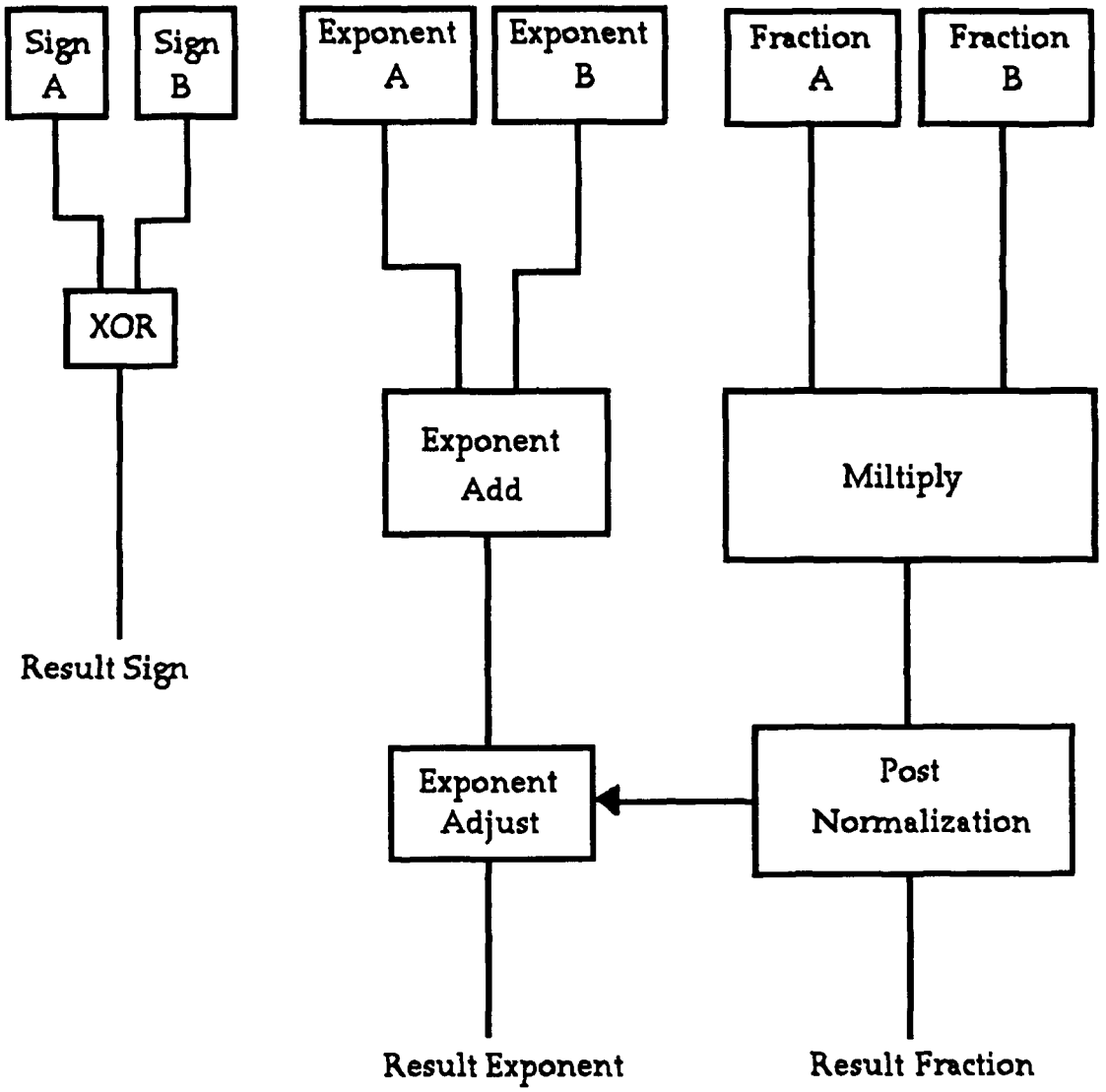


Figure 2.2 Block Diagram for Floating Point Multiply

"exclusive-or" of the sign bit of the operands. Note that the above three operations can be performed simultaneously; they do not depend on each other.

In any floating-point operation, the normalization process is necessary for producing the correct result. It has been observed that the product of two normalized floating-point numbers may have one of the three possibilities in the value left to the decimal point: 11, 10, 01. This can be shown as follows.

$$
\begin{array}{rl}
 & 1.\ \mathrm{AAAA}..... \\
\times) & 1.\ \mathrm{BBBBB}..... \\
\hline
 & 11.\ \mathrm{fffff}..... \\
\text{or} & 10.\ \mathrm{fffff}..... \\
\text{or} & 01.\ \mathrm{fffff}.....
\end{array}
$$

The result of either 11 or 10 indicates an overflow due to a carry and the product must be normalized (shifted to the right one bit). There is no need to normalize a product if the result is 01 (i.e., no carry has occurred). Note that when the normalization takes place the value in the exponent field must be adjusted to reflect the correct value. For example, after normalization, the value $(11.1000)_2 \times 2^3$ is equal to $(1.1100)_2 \times 2^4$.

2. Floating-Point Addition

Compared to the multiplication, floating-point addition is a much more complex operation. This is because that addition requires both numbers

to have the same exponent, in order to have proper alignment of the fractional portion of the number. The sign bit also must be tested to ensure the proper result, and not just an absolute value determination made. Figure 2.3 shows a block diagram for the addition process [Ref. 6].

Before the two floating-point numbers can be properly added together, the fraction must be aligned (unless the exponents of the two numbers are the same). This involves determining which operand is smaller, and then aligning the fraction of that operand appropriately with the fraction of the larger operand. The alignment is accomplished by shifting the fraction of the smaller operand a number of positions to the right, therefore making the digits of the smaller operand line up with the digits of the same significance in the larger operand. The amount of the alignment, the number of positions to shift, is determined by the difference in the exponents. The addition element then receives the fraction directly from the larger operand, and the aligned fraction from the smaller operand. The resulting number is then provided to the POST NORMALIZATION unit. The post normalization unit must be capable of a shift of at least one position to lesser significance, and must also be capable of shifts of many positions to higher significance. Note that this post normalization must then be capable of adjusting the size of the exponent to reflect any normalization. At the end of this process, the result will have been properly formed and ready for any additional operation required of it.[Ref. 6]

The floating-point addition also includes subtraction, since the sign-magnitude method of storing information necessitates that the hardware be capable of both.[Ref. 6]

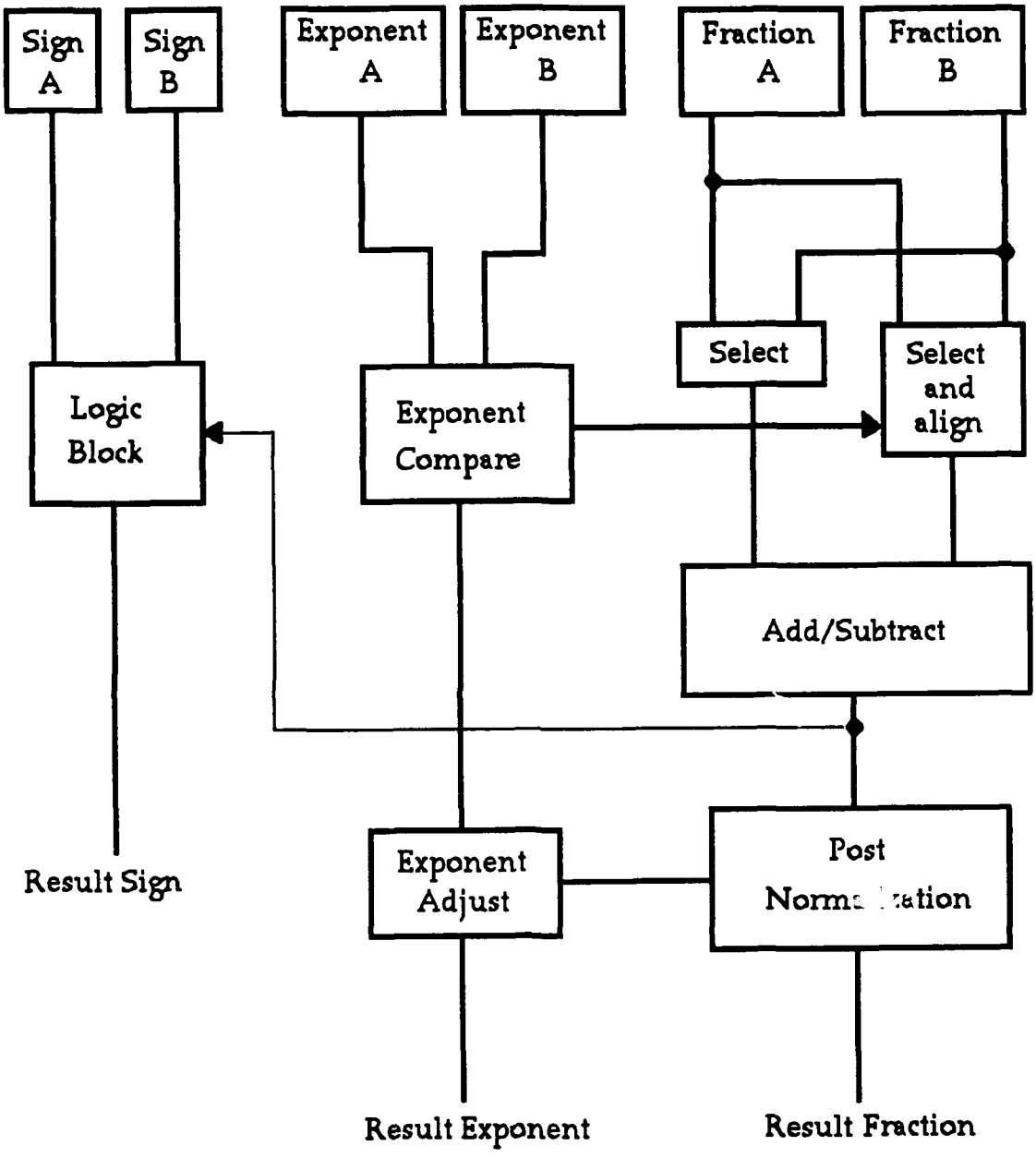


Figure 2.3 Block Diagram for Floating Point Addition

3. Overflow, Underflow, and Extra bits

Overflow occurs when an operation produces a result that has exceeded the ability of the system to represent information. This can occur when multiplying two numbers whose exponents sum up to an exponent not representable in the system, or adding two numbers that already are the maximum representable numbers by the system. Overflow can also be caused by a post normalization (e.g., when the exponent is 1111 and after normalization has occurred, the resulting exponent is 0000 which does not represent the value desired.

Underflow occurs when an operation produces a result that is too small to represented in the number system. This can occur in the multiplication process when two negative exponents are added and the resulting exponent can not be represented in the system (e.g., if one exponent is 0001 (2^{-7}) and the other exponent is 0100 (2^{-4}), then the resulting exponent is 2^{-11} that can not be represented in this number system).

The floating-point operations of multiplication, addition, and subtraction may increase the number of bits in the fraction beyond the number we can actually store. There are many ways to deal with these *extra bits*. The obvious and simplest method is merely to ignore them; this is called truncation. The unwanted bits are dropped from the result. This results in a bias; the magnitude of the number to be stored is smaller than the true value. The VLSI cells developed in this thesis use the simple truncation technique. The designs are easily modified for rounding or jamming. Other more complex techniques (e.g., zero bias rounding and ROM rounding [Ref.

6]) can be used with more modifications, but increase delay of the unit and increase its size.

C. FFT STRUCTURE

For a Discrete Fourier Transform (DFT) calculation, the operations are performed on a sequence of data. Assume that the total number of input data is N, which is a power of two. For a finite-duration sequence x(n), the DFT formula is defined as:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi nk/N} \qquad \text{for} \quad k = 0, 1, \ldots, N-1$$

The FFT terminology refers to methods for fast computation of the DFT [Ref. 7]. In the following, a brief description of two data flow designs of Fast Fourier Transform are presented. They are the methods of Decimation In Time and Decimation In Frequency [Ref. 7].

1. Decimation In Time (DIT)

This method divides x(n) into two halves: one with odd sequence numbers, and the other with even sequence numbers Through the well-known butterfly operation graph for the DIT, the fast fourier transform can be represented by Figure 2.4. Where W^k is known as the *weighting factor*, and is related to the $e^{-j2\pi nk/N}$ term in the DFT.

2. Decimation In Frequency (DIF)

Another equivalent way to decompose the calculation of the DFT is known as the decimation in frequency. The signal flow of the butterfly is shown in Figure 2.5.

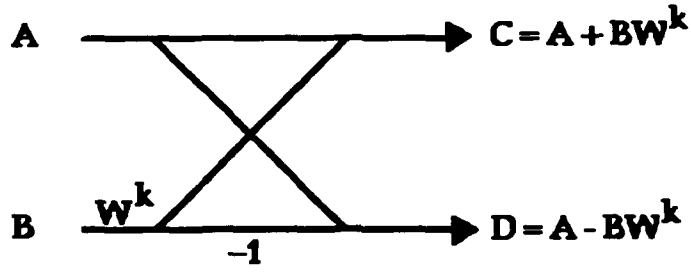


Figure 2.4 Signal Flow Graph of DIT Butterfly

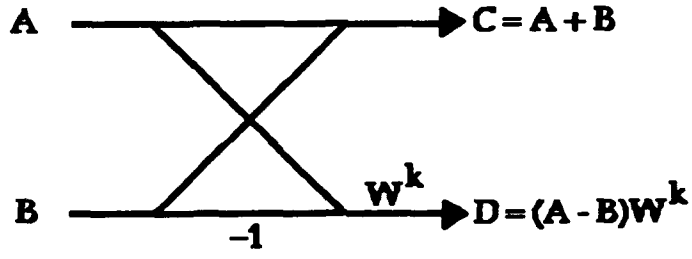


Figure 2.5 Signal Flow Graph of DIF Butterfly

A radix-2 FFT (two-point DFT), either DIT or DIF requires one complex multiply and two complex additions. These are equivalent to four multiplications and six additions of real numbers.

III. THE HARDWARE DESIGN OF FLOATING-POINT MULTIPLY UNIT AND ADD UNIT

The algorithm for floating-point multiplication and addition has been described in Chapter II. The hardware configuration of the floating-point MULTIPLY unit and ADD unit is described in section A and section B. These two units can be used as the basic hardware components for illustrating the execution of standard floating-point operations. The hardware of the floating-point SUBTRACT unit will be discussed in section C, and the the hardware of the floating-point FFT unit will be presented in section D.

A. THE HARDWARE DESIGN OF FLOATING-POINT MULTIPLY UNIT

The floating-point multiply unit includes adders, a parallel multiplier, multiplexers (MUXs), and logic gates. Figure 3.1. shows the block diagram of floating-point multiply unit. The two input operands to be specified as *A* and *B*, and the result will be W. Thus, within the floating-point unit, the operand *A* will be made up of the catenating of its three components,

$$A = as, ae, am$$

the 1 bit sign, *as*, the 4-bit exponent, *ae*, and the 11-bit fraction, *am*.

There are seven major steps which must be executed in order to complete the multiplication of two 16 bit floating-point numbers.

1. Check for zero operands

As we mentioned in Chapter II that the floating-point format is the IEEE floating-point standard and scaled down to 16 bits, and the exponent is a excess-8 number represented using radix-2. With this format the smallest

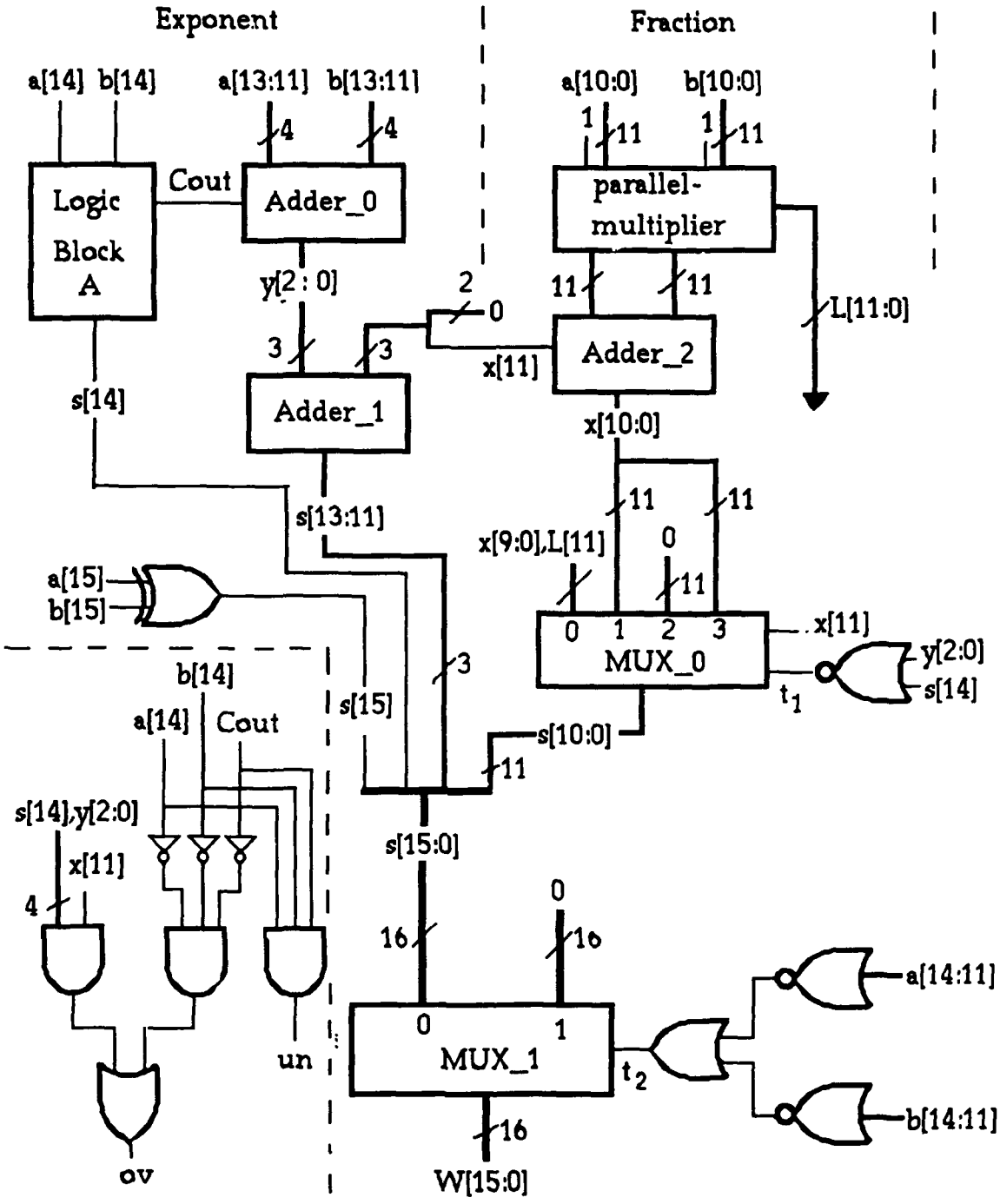


Figure 3.1 The Block Diagram of Floating-point Multiply Unit

number that can be represented is $(1.000...)_2 \times 2^{-7}$. Thus, a number that is between $[-V_{min} \sim +V_{min}]$ will be treated as zero. Therefore, when the exponent of either the multiplier or the multiplicand is 2^{-8} (i.e., 0000) the product should be zero. To test if one or both of the operands are zero, we can use two NOR gates and one OR gate to implement. This testing result will be used to select the final product, which is shown in Figure 3.1.

2. Set the product sign

In our floating-point number system, a negative number has its sign bit as 1. The sign portion of a floating-point multiplier can be produced with an XOR gate. The sign bit is a "1" only when the signs of the multiplier and the multiplicand are different.

3. Exponent addition

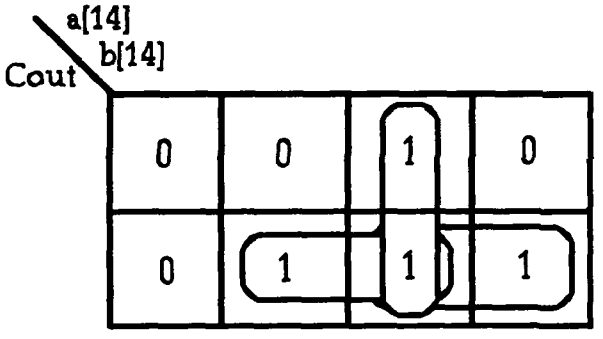
Refer to Figure 3.1, we can accomplish the exponent addition with the logic block A and two adders. Having observed in Table 2.1 on page 5, the excess-8 number representation, we notice that the most significant bit (MSB) in the exponent addition can be represented as:

$$S[14] = a[14]b[14] + a[14]C_{out} + b[14]C_{out}$$

This can be explained as follows. In Figure 3.1 C_{out} is the carry-out of the Adder_0. There are three possibilities when adding two exponents: both positive, both negative, and opposite in signs. Refer to Table 2.1, the MSB of the exponent of an excess-8 number is 1 (0) when it is a positive (negative). Therefore, when both a[14] and b[14] are positive (negative), the sign bit should be 1, when a[14] and b[14] are opposite in signs the resulting sign depends on the C_{out}. The truth table, Karnaugh map, and the gate structure are shown in the Figure 3.2.

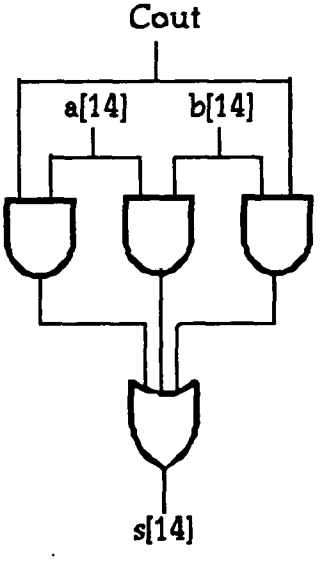
a[14]	b[14]	Cout	s[14]
0	0	x	0
0	1	0,1	0,1
1	0	0,1	0,1
1	1	x	1

(a) Truth Table



s[14] = a[14]b[14] + a[14]Cout + b[14]Cout

(b) K map



(c) Gate structure

Figure 3.2 The Logic Design of Block A in MULTIPLY Unit

4. Fraction multiplication

The fraction multiplication can be implemented by one parallel multiplier and one adder (Adder_2). Note that the MSBs of the two input operands of the parallel multiplier are the hidden ones.

5. Normalization

As described in Chapter II, the fraction multiplication may have three possible integer parts (two digits to the left of the binary point): 11, 10, and 01. If the integer part is 11 or 10 (i.e., x[11] = 1), we normalize by taking 11 bits (x[10 : 0]) as the significant and increment the exponent sum by 1. If the integer part is 01, we simply discard the leading 0 of the significant and take the 11 bits (x[9 : 0], L[11]) as the fraction output [Ref. 8]. Note that the hidden 1 (x[11] or x[10]) is implied.

6. Force to zero

Figure 2.1 on page 7 shows the range of zero. The normalization, after significant multiplication, could adjust the product exponent into the representable range (e.g., the product exponent of 0100 (i.e., 2^{-4}) and 0100 (i.e., 2^{-4}) is 0000 (i.e., 2^{-8}), and the integer part of the product fraction is 11 or 10; after the normalization, the product exponent is adjusted to 0001 (i.e., 2^{-7})). In the other words, the result of exponent addition is 0000 (i.e., 2^{-8}) and the carry-out (x[11]) of fraction multiplication is zero that will force the result of this multiplication to zero. Figure 3.1 shows the output of the NOR gate and bit x[11] select the output (S[10 : 0]) of MUX_0 is zero, shifting the significant one place to the right, or simply discard the leading 0 of the significant.

7. Overflow and underflow

Using Table 2.1 and Figure 2.1, we know that overflow can occur when a[14], b[14] and C_{out} are ones. Since the shifting for normalization can never be more than one digit, so overflow can also occur when the result of exponent addition is 1111 (i.e., 2^{+7}) and the carry-out, x[11], of Adder_2 is "1" (i.e., 2^{+8} can not be represented in Figure 2.1). Underflow can occur when a[14], b[14] and C_{out} are zeros. Note that a[14] and b[14] are the MSBs of the exponent part of two operands. Three AND gates, three inverters, and one OR gate show the overflow flag (ov) and the underflow flag (un) in Figure 3.1.

B. THE HARDWARE DESIGN OF FLOATING-POINT ADD UNIT

The floating-point ADD unit includes adders, multiplexers, programmable logic array (PLA), shifter, and logic gates. Figure 3.3 shows the logic blocks diagram of floating-point ADD unit. As described in Chapter II, the floating-point addition is more complicated than the floating-point multiplication. The reason for this is the alignment of operands required before initiating the floating-point addition and normalization is required at the end of the transaction.

There are six major steps which must be executed in order to complete the addition of two 16 bit floating-point numbers.

1. Align the fraction by equalizing their exponents

The comparison of the exponents is realized with two's complement subtraction. Therefore, a two's complement converter is needed, and shown in Figure 3.4 [Ref. 12]. Two adders (Adder_0, Adder_1) compare the two exponents, *ae*, and *be*. There are three possibilities, $ae > be$, $ae = be$, and $ae < be$. If the result is either $ae > be$ (i.e., Ae[3] = 0) or $ae < be$ (i.e., Ae[3] = 1), then

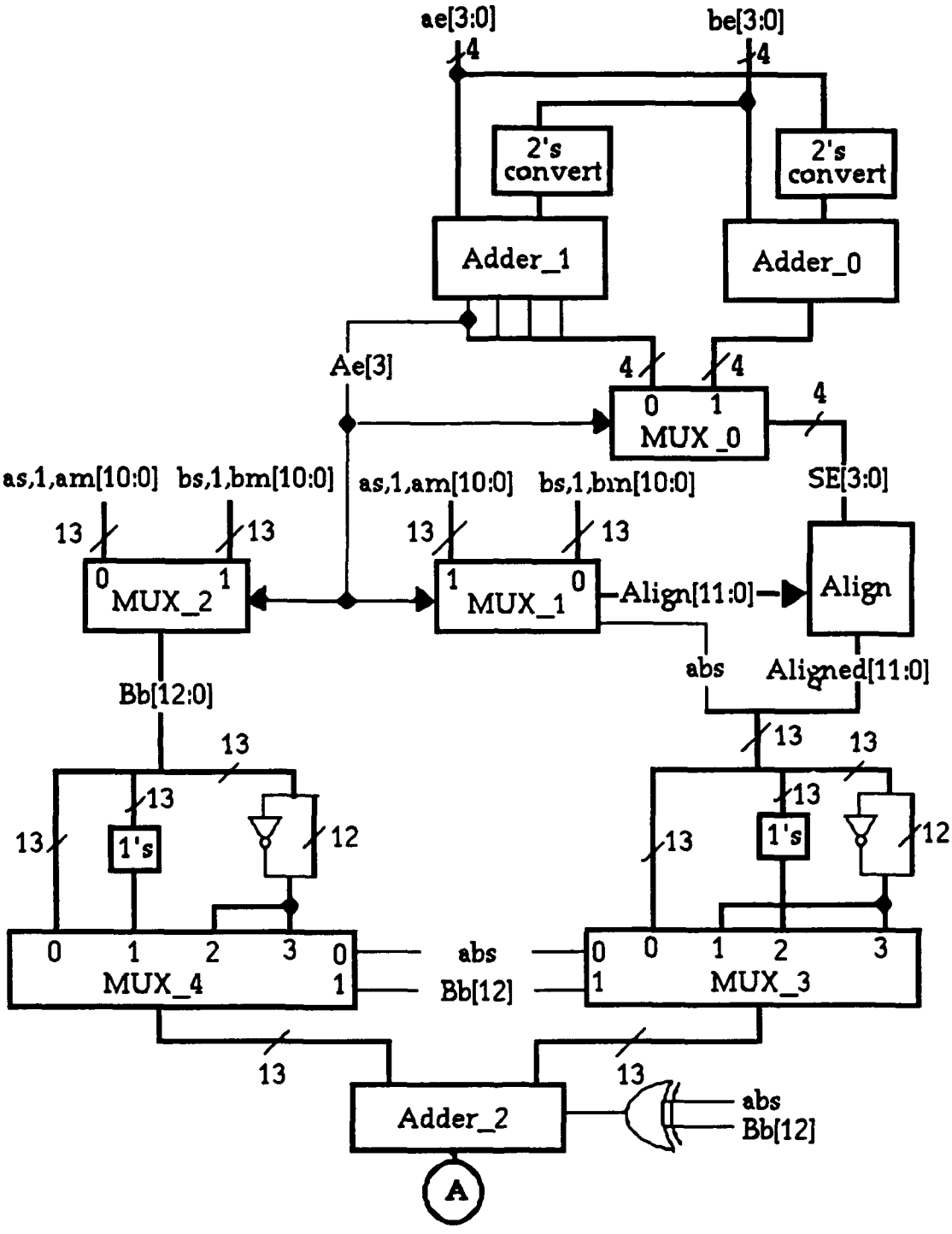


Figure 3.3.a The Block Diagram of Floating-point Add Unit.

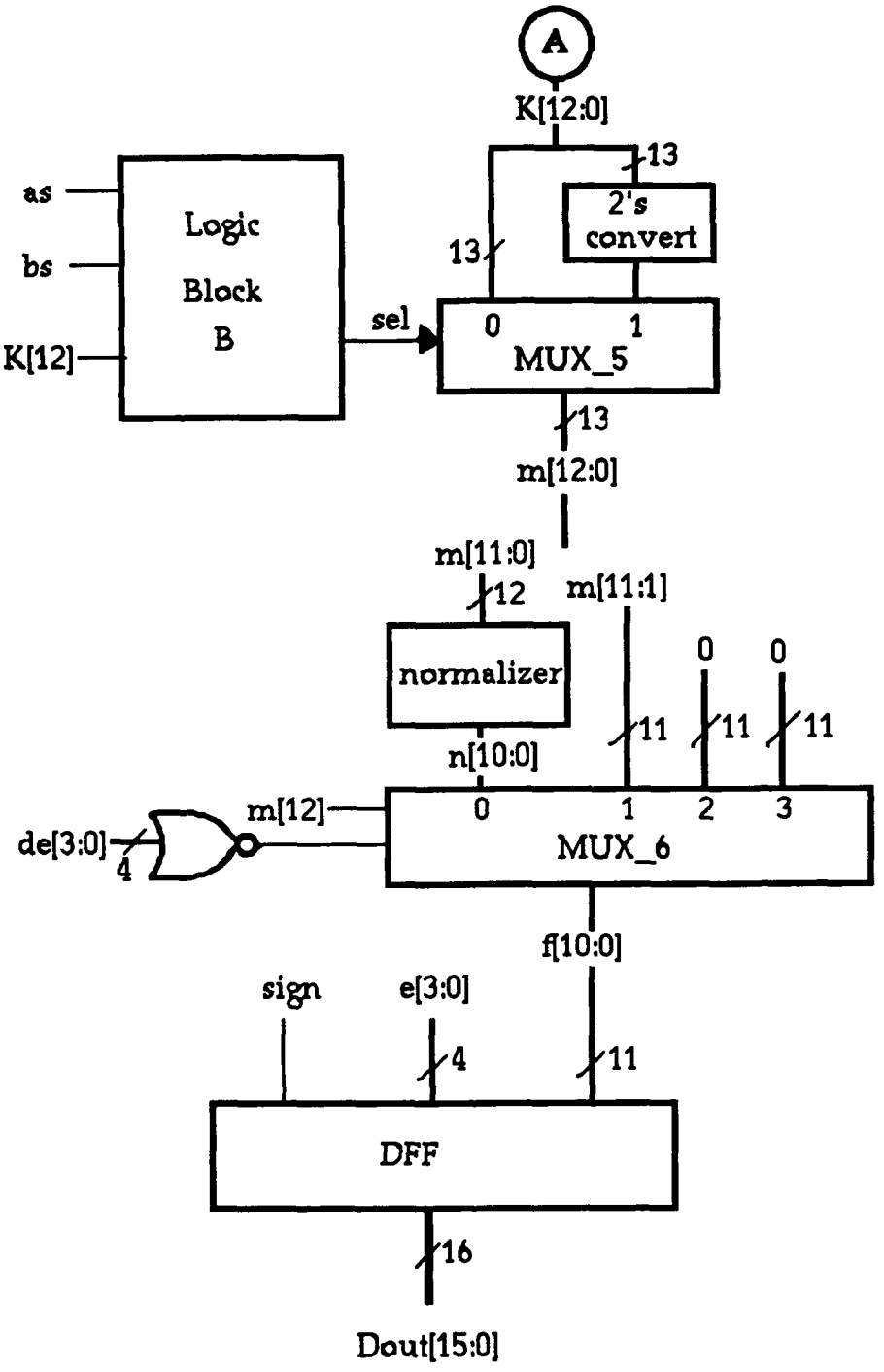


Figure 3.3.b The Block Diagram of Floating-point Add Unit.

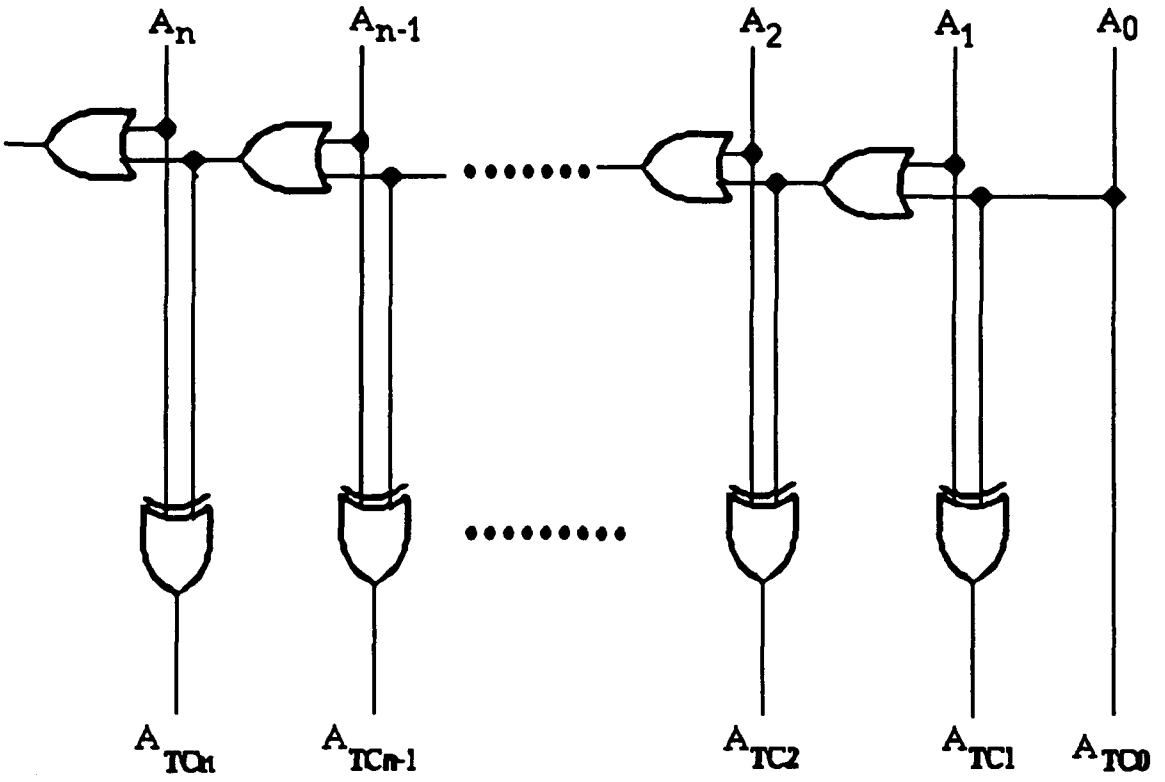


Figure 3.4 Two's Complement Converter

the smaller exponent must be incremented to match the magnitude of the larger exponent. Since the number system is based on two, the alignment network must be capable of shifting any number of bits, from zero to twelve. Comparison of the two exponents provides a binary number (SE[3 : 0] in Figure 3.3.a) which indicates how far the smaller exponent needs to be shifted to complete the alignment process. Design of the network used to align the smaller fraction to be added to the larger fraction is in Figure 2.3 on page 11. Figure 3.5 shows one way of accomplishing this is to use four 2-1

multiplexers. The MSB (SE[3]) of this number is then used by the first level of MUXs to shift the number by eight bits (the 1 condition), or provide no shift at all (the 0 condition). Similarly, the second MSB (SE[2]) of the number is used by the second set of MUXs to shift the number provided by the first set of MUXs by four (the 1 condition) or provide no shift at all (the 0 condition). This process continues, with each level of multiplexers shifting the number by some power of two, until all four bits have been aligned. For example, if SE[3 : 0] = 0110, then the smaller fraction will be shifted to the right 6 bits.

2. Add/subtract the fraction

The result of the comparison (*ae* and *be*) directs the MUX_2 to select the unaligned fraction (larger exponent), and the same signal directs the MUX_1 to select the other fraction (smaller exponent) and align it by shifting the appropriate number of positions. These two results, one unaligned fraction and one aligned fraction, are then fed to the Adder_2 for the actual calculation. Since both addition and subtraction must be accommodated, the use of two's complement arithmetic is appropriate. Which will require conversion of operands from sign-magnitude to two's complement form. For either addition or subtraction, a negative operand (an addend or the minuend) is converted to two's complement form by complementing the significant. For addition, the second operand is similarly complemented if it is negative. For a negative number, all that is needed is to change the sign bit; for a positive number, both the sign bit and the significant must be complemented. The two sign bits (abs, Bb[12]) select this operation for true addition or true subtraction. Note that the ones' converter and the XOR gate

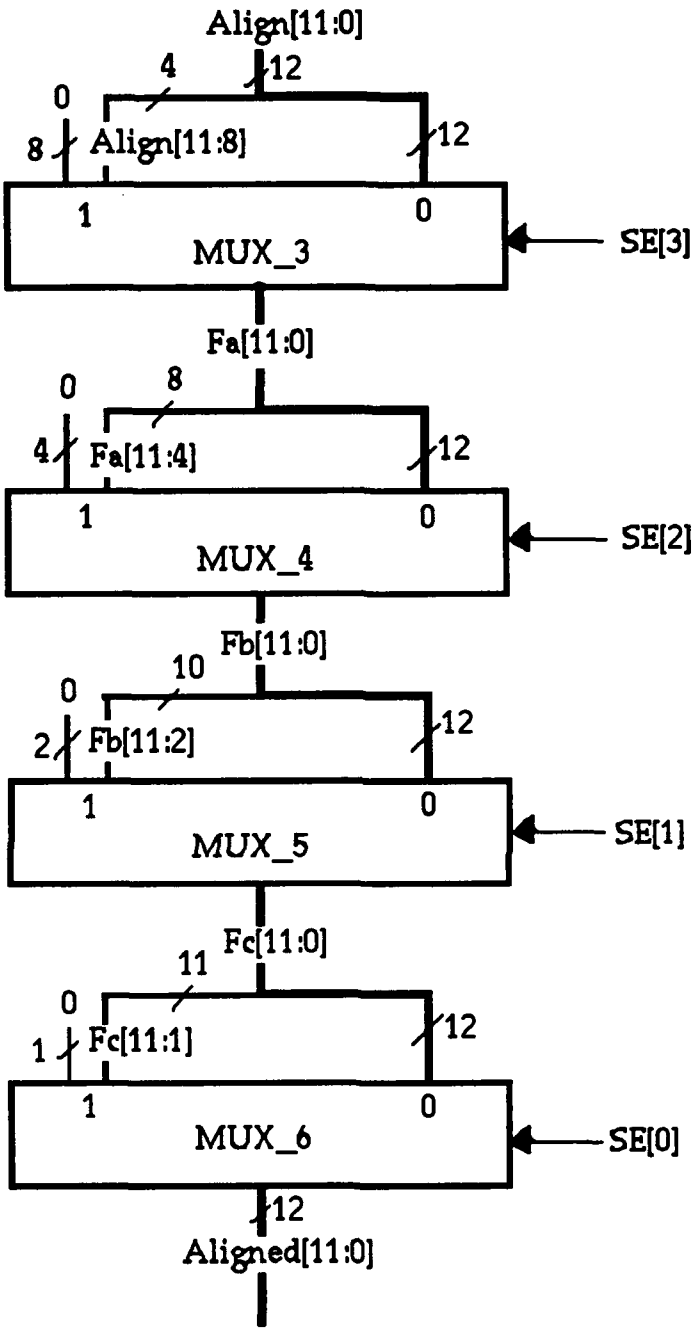


Figure 3.5 Logic for Alignment Shift Network

replace the two's converter.

The above statement on two fractions (including the sign bit and hidden 1) addition indicates, when the result is negative, that in two's complement form should be recomplemented back to its sign-magnitude form. The Logic Block B consists of two AND gates, two inverters, and one OR gate, its truth table, K map, and the gate structure are shown in Figure 3.6. The output (signal : sel) of this logic block B directs the MUX _5 to select the fraction sum of two input operands, and the nonzero result will be normalized by an programmable logic array (PLA) and a shifter (as shown in the Figure 3.7 on page 28).

3. **Normalize the resulting sum/difference**

The fraction sum/difference may result in four possible cases as follows:

case 1: m[12 : 0] = 10XXX.....

case 2: m[12 : 0] = 11XXX.....

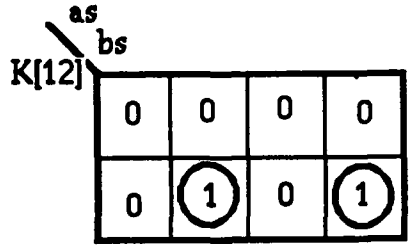
case 3: m[12 : 0] = 01XXX.....

case 4: m[12 : 0] = 00XXX.....

Where X = 0 or 1. A check is made to see if m[12] = 1; this is actually done by shifting the fraction one bit right, and adding one to the result exponent, i.e., case 1 and case 2. Note that the least significant bit being lost due to truncation. If not, the result is shifted left until a nonzero digit appears in the MSB (i.e., m[11]) position, and decreasing the resultant exponent by one for each shift, i.e., case 4. The normalization is implemented by an PLA and a shifter. PLAs are implemented as two-level logic realization of a sum-of-

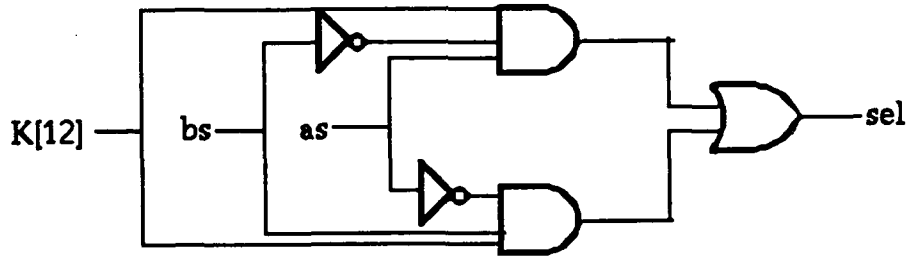
as	bs	K[12]	sel
0	0	x	0
0	1	0,1	0,1
1	0	0,1	0,1
1	1	x	0

(a) Truth table



$sel = \overline{as}K[12]bs + asK[12]\overline{bs}$

(b) K map



(c) Gate structure

Figure 3.6 The Logic Design of Block B in ADD Unit

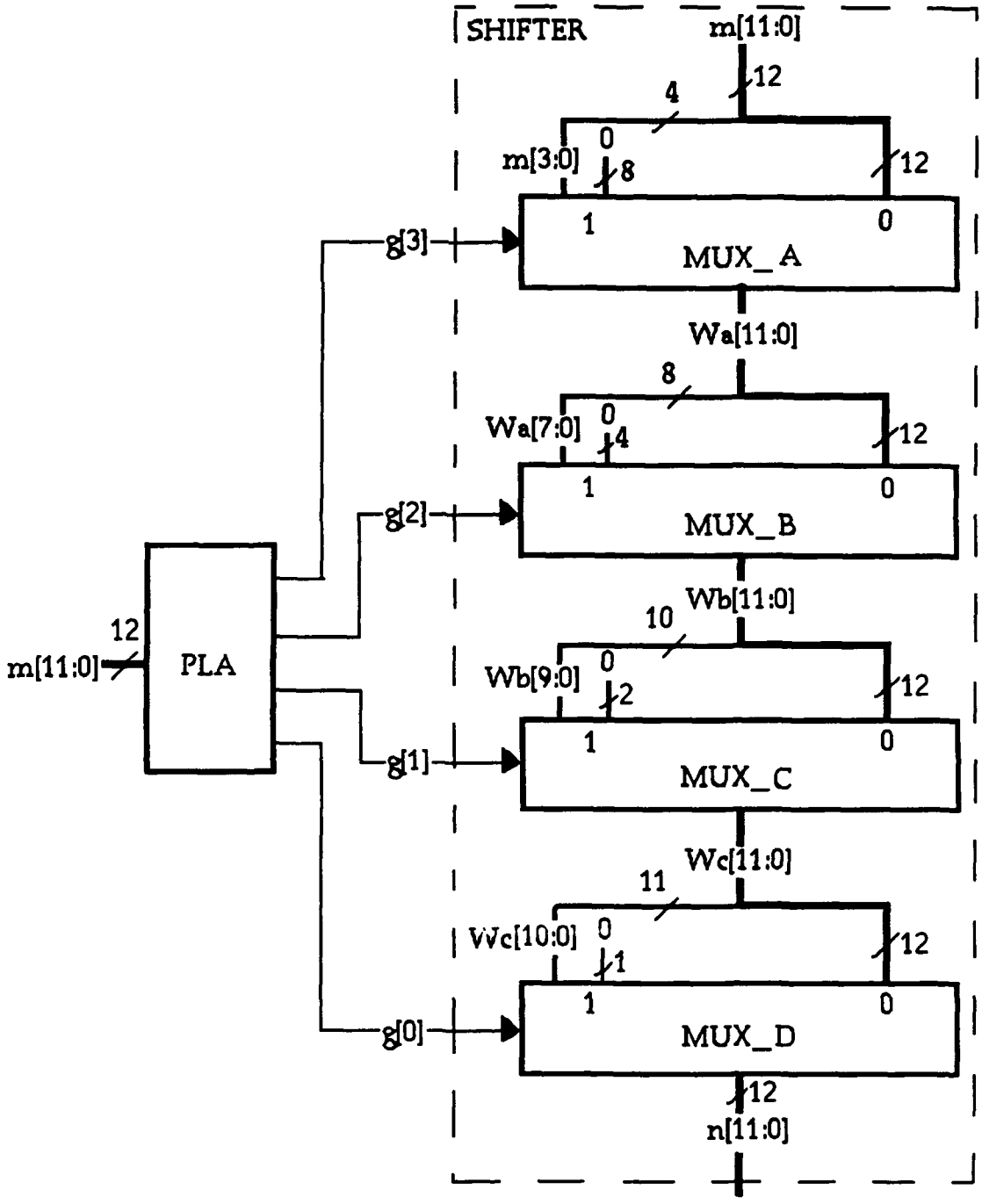


Figure 3.7 The Normalizer of Floating-point ADD Unit

products expression. The output signals can be expressed as the sum (OR) of several intermediate signals, each of which can be expressed as the product (AND) of several input signals. We use a readily available PLA that is in the library of the Genesil Silicon Compiler. The equations that determine the logic of the PLA are written in PLAEQ [Ref. 17], the PLA specification language, and contained in an ancillary file. This file is parsed by the PLA parser and can be optimized by the optimizer of choice. The Compiler uses this parsed and optimized file to generate the layout, simulation, and timing models [Ref. 17]. The Ancillary File parameter is used to name the PLA ancillary file that contains PLAEQ coding. This coding specifies the input and output signals and their attributes and the logic to be implemented by the PLA. In the normalization of the floating-point addition, how far the bit is needed to shift to left that is implemented by the PLA ancillary file which is shown in below.

```
**************************************************************************
CODEFILE

    INPUT m[11];
    INPUT m[10];
    INPUT m[9];
    INPUT m[8];
    INPUT m[7];
    INPUT m[6];
    INPUT m[5];
    INPUT m[4];
    INPUT m[3];
    INPUT m[2];
    INPUT m[1];
    INPUT m[0];
    OUTPUT g[3];
```

```
    OUTPUT g[2];
    OUTPUT g[1];
    OUTPUT g[0];

CODING (ROM)

    < 1 . . . . . > 0000;
    < 01 . . . . . > 0001;
    < 001 . . . . . > 0010;
    < 0001 . . . . . > 0011;
    < 00001 . . . . . > 0100;
    < 000001 . . . . . > 0101;
    < 0000001 . . . . . > 0110;
    < 00000001 . . . . > 0111;
    < 000000001 . . . > 1000;
    < 0000000001 . . > 1001;
    < 00000000001 . > 1010;
    < 000000000001 > 1011;
    < 000000000000 > 1100;

END
```

****

Design the shifter used to shift the fraction bit to the left (i.e., normalization) that is very similar to the alignment shift network, which has described in Figure 3.5 on page 25, except the shifter shift bit to the left.

Note that two M-bit fraction numbers, when subtracted, may result in a required post normalization alignment of M-1 positions [Ref. 6]. In the event of m[12 : 11] = 01, the post normalization step is skipped, i.e., case 3. Step 3 is shown in the Figure 3.3.

4. Adjusting the exponent

As step 1 described, the signal (Ae[3]) directs the MUX_9 to select the larger exponent. In step 3, the exponent result may be increased one,

decreased one to eleven, or no change. Note that the decreasing operation is executed by a two's complement convert.

After the normalization, if the adjusted exponent result, de[3 : 0], is 0000 and the resultant fraction sum/difference is case 3 then the result of this floating-point addition is *zero* as far as our resolution can determine (as shown in the Figure 2.1 on page 7). If the resultant fraction sum/difference is m[12 : 0] then this indicates that the floating-point addition has resulted in a value that is a *true zero* [Ref. 11]. Note that, after adjusting the exponent, the result exponent (e[3 : 0]) will be selected. Step 4 is shown in Figure 3.8.

5. Exponent overflow and underflow

Overflow occurs whenever one of the incoming exponents is 1111 (the maximum value) and the result of the addition causes the exponent to be incremented. This sets the exponent to 0000 which is an indication of overflow.

Underflow can occur only when the normalization was executed as described in step 3. Figure 3.8 shows the m[12 : 11] = 00 ensure the left shifting of fraction has happened, and the carry-out, C_{out}, of Adder_3 is zero to indicate the exponent exceeds the limit during true addition. For example, assume the larger exponent is 0001 (i.e., 2^{-7}), m[12 : 0] = 0001000000000, and C_{out} = 0. After the left shifting of fraction, the resulting exponent, 2^{-9}, exceeds the limit and shows that the exponent underflow has occurred.

6. Setting of the sign bit

As described in step 2 and step 3, the resultant sign bit of floating-point addition is decided by *as*, *bs*, and *m*[12]. Figure 3.9 shows the truth table, K map, and the gate structure of this logic.

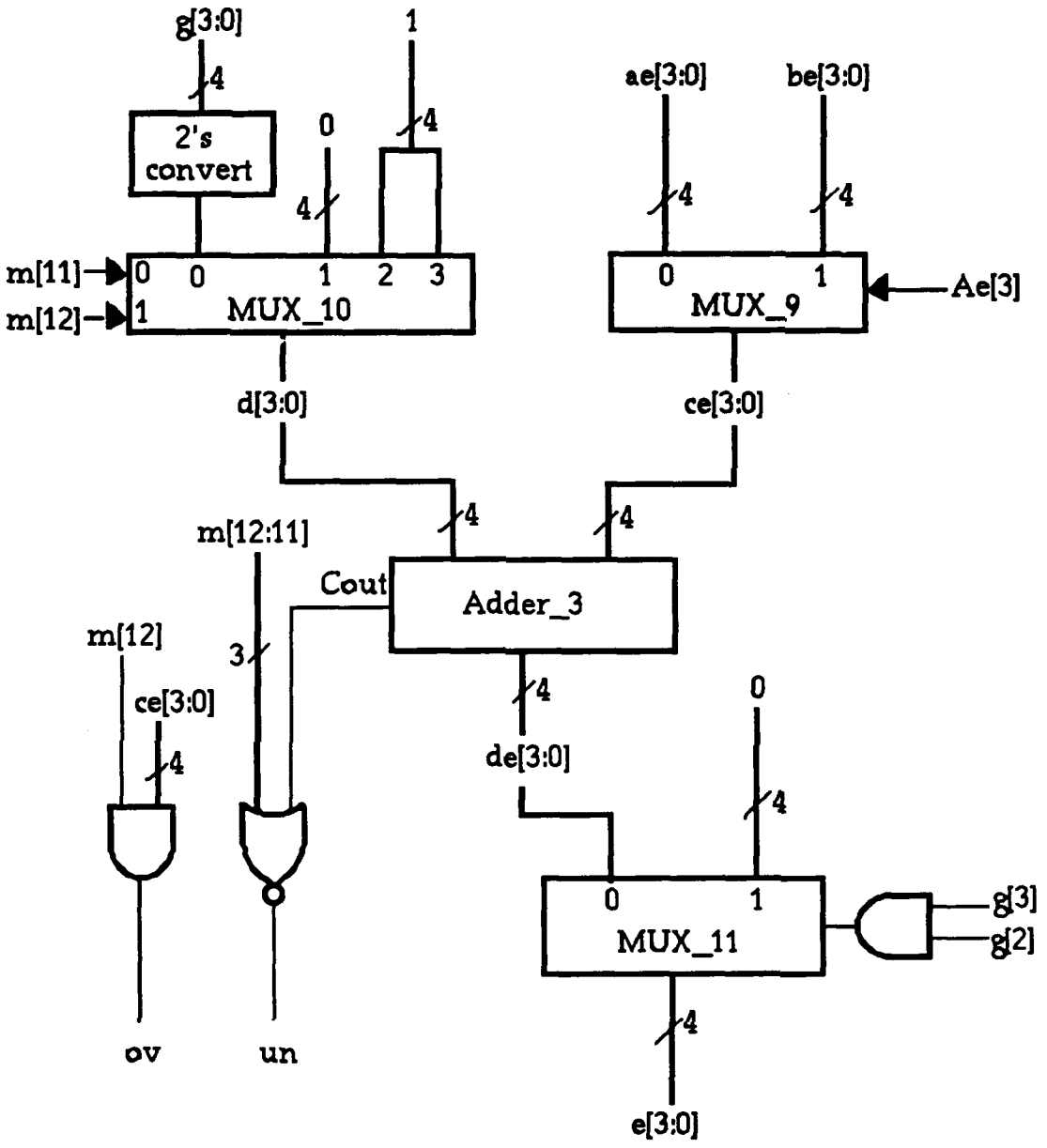
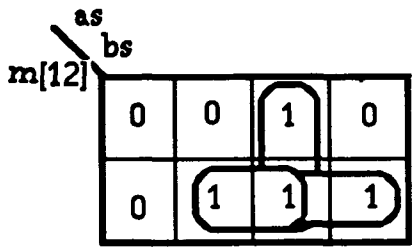


Figure 3.8 The Underflow, Overflow, and Exponent Operation of Floating-point ADD Unit

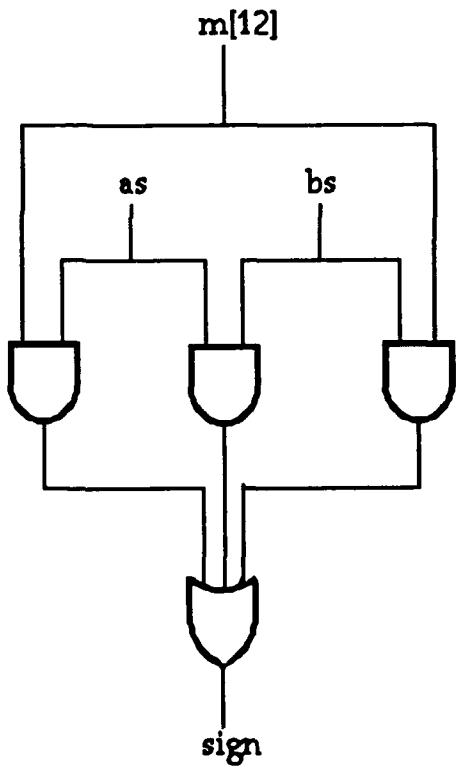
as	bs	m[12]	sign
0	0	x	0
0	1	0,1	0,1
1	0	0,1	0,1
1	1	x	1

(a) Truth table



sign = asbs + m[12]as + m[12]bs

(b) K map



(c) Gate structure

Figure 3.9 The Logic Design for Sign Bit in ADD Unit

C. THE HARDWARE DESIGN OF FLOATING-POINT SUBTRACT UNIT

The only difference between floating-point addition and floating-point subtraction is that the sign bit of the second operand (augend/minuend) will be reversed (i.e., A + (–B) = A – (+B) or A – (–B) = A + (+B)). Note that the difference in the hardware configuration is an inverter that is set in front of the sign bit of the second operand (i.e., B).

D. THE HARDWARE DESIGN OF FLOATING-POINT FFT

As previously described in Chapter II, the hardware configuration of floating-point FFT (i.e., two-point DFT with complex number input) is composed of one complex ADD (two real adder), one complex SUBTRACT (two real subtracter), and one complex MULTIPLY units. The complex multiply consists of four real multiply operations and two real add operations. In this thesis, the simplest floating-point FFT (i.e., two-point DFT with complex number input) has been implemented. The block diagram of this floating-point FFT is shown in Figure 3.10. The two-point DFT is shown in Figure 2.4 and Figure 2.5, which is the basic processor for layer FFTs, is known as the *butterfly*. For a detailed description of N-point, where N is a power of two, floating-point FFT the reader is refered to Reference 7.

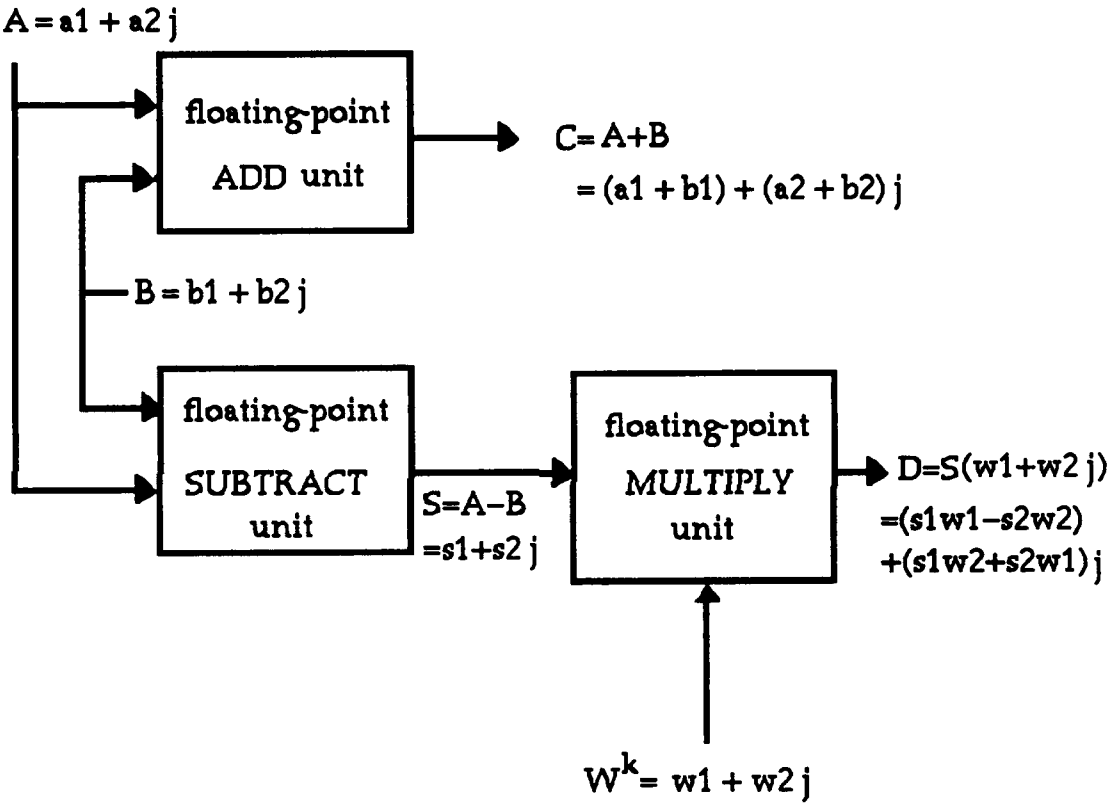


Figure 3.10 The Block Diagram of Floating-point FFT

IV. DESIGN VERIFICATION

The GENESIL can simulate a digital design at both the functional and switch-level to verify design functionality. Using GENESIL, the user specifies the design functionality and the netlist, and builds the functional simulation models. The simulator uses these models, which contain initialization conditions and additional simulation commands, to verify the operation of the design. The Simulator can be controlled directly on an interactive screen interface or run in batch mode. Figure 4.1 illustrates the Simulator environment within GENESIL [Ref. 13-15]. A GENESIL design (i.e., floating-point ADD, SUBTRACT, MULTIPLY, and FFT units are described in Chapter III) is specified. The simulation is performed on functional models which derived from the block specifications and netlists [Ref. 15].

Four different simulation models (GFL, FLATGFL, FLATSGFL, and GSL) are available on the GENESIL Simulation Menu [Ref. 15]. The GFL functional model is a gate-level model used for general-purpose simulation. This is the simulation model utilized for this thesis. With the GFL model, all simulation nodes are available, and the design hierarchy is preserved for use by Simulator commands. For example, the "pi" command will list the inputs and outputs to the selected instance of the block, module, chip, or chipset [Ref. 15].

In the following sections, the functional simulation results of floating-point MULTIPLY and ADD unit will be described by examples. Note that some special cases are also specified.

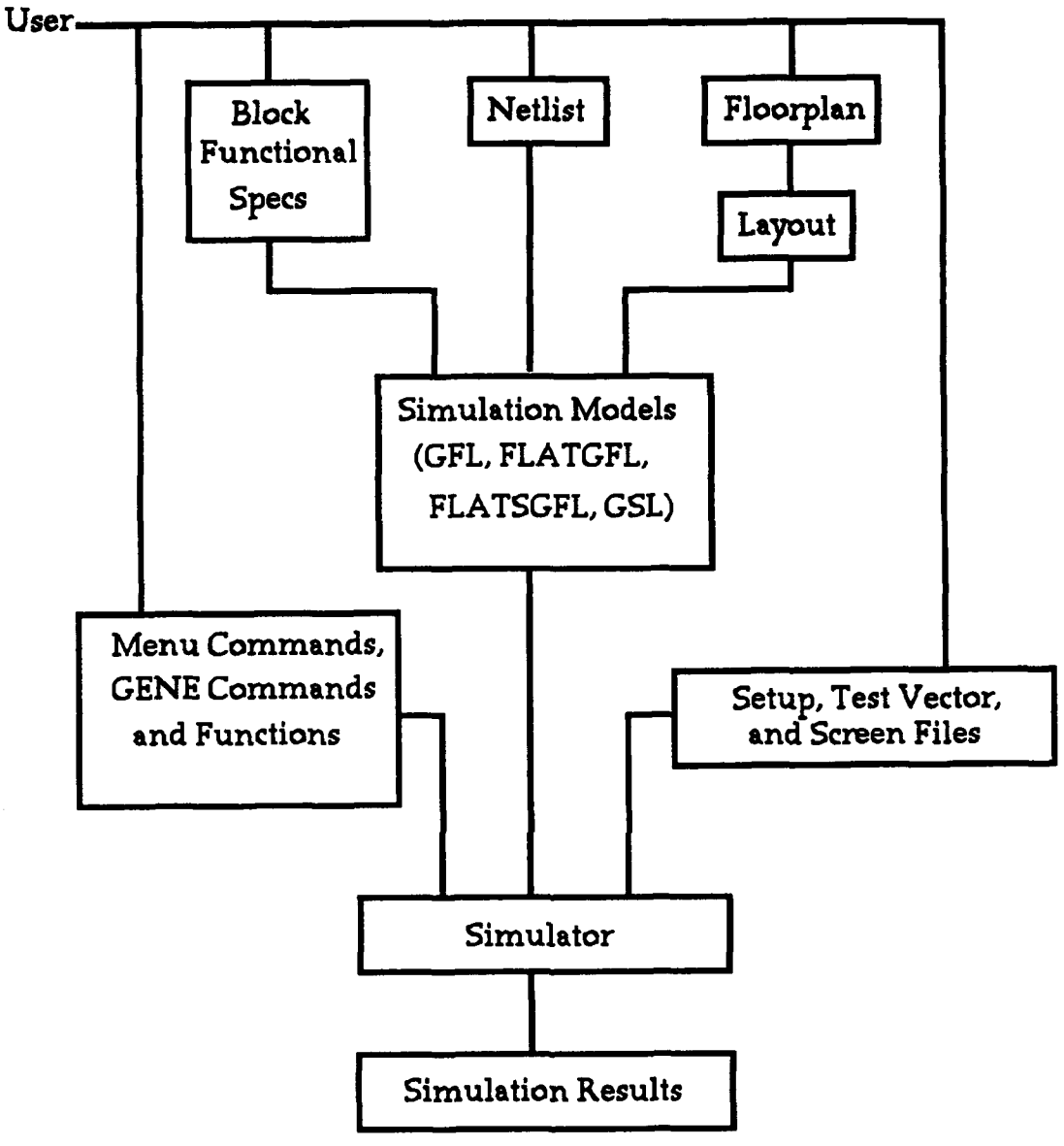


Figure 4.1 The Simulation Environment within GENESIL

A. THE SIMULATION RESULT OF FLOATING-POINT MULTIPLY UNIT

Example 1:

The floating-point format, 7.0 can be represented as:

$$7.0 = 0\ 1010\ 11000000000$$

Since 7.0 is a positive number and the sign bit is 0. The exponent is in excess-8 so that 1010 represents 2. The fraction is 1.11 = 1 (hidden 1) + 2^{-1} + 2^{-2} = 1 + 0.5 + 0.25 = 1.75. Therefore this expression is for a value of $1.75 \times 2^2 = 7.0$. When simulation is executed on the multiplication of 7.0 by 7.0, the simulation result obtained from Figure 4.2 is

$$W[15:0] = 0\ 1101\ 10001000000$$

that can be verified to be the value of $1.53125 \times 2^5 = 49$.

Note that x[11] = 1 illustrates that normalization has occurred. Both ov = 0 and un = 0 indicate that neither overflow nor underflow has occurred. The correct result shows no truncation error in this example.

Example 2:

Let A be the multiplier, and B be the multiplicand. The values for A and B are as follows (both in floating-point format and decimal format):

$$A = (0\ 1010\ 11000100000)_2 = (1.765625 \times 2^2)_{10}$$

$$B = (1\ 1010\ 11001000000)_2 = (1.78125 \times 2^2)_{10}.$$

Simulation in GFL gives the result in Figure 4.3 as:

$$W[15:0] = 1\ 1101\ 10010010100$$

that can be verified to the value of $-1.572265625 \times 2 = -50.3125$. Note that the correct product of A multiplied by B is -50.3203125. The difference value, 0.0078125 ($= 50.3203125 - 50.3125$), is due to truncation error.

Example 3:

Let $A = 1\ 1100\ 11000000000 = (-1.75 \times 2^4)_{10}$

$B = 1\ 1100\ 11000000000 = (-1.75 \times 2^4)_{10}$,

the correct product of A multiplied by B is $(784)_{10}$.

Simulation in GFL gives the result in Figure 4.4 as:

$W[15 : 0] = 0\ 1001\ 10001000000 = (3.0625)_{10}$

The ov = 1 indicates an exponent overflow has occurred and therefore the value in W[15 : 0] is useless.

Example 4:

Let $A = 1\ 1100\ 11000000000 = (-1.75 \times 2^4)_{10}$

$B = 1\ 1011\ 11000000000 = (-1.75 \times 2^3)_{10}$,

the correct product of A multiplied by B is $(392)_{10}$.

Simulation in GFL gives the result in Figure 4.5 as:

$W[15 : 0] = 0\ 1000\ 10001000000 = (1.53125)_{10}$.

The ov = 1 indicates an exponent overflow, which is caused by normalization, has occurred and therefore the value in W[15 : 0] is useless.

Example 5:

Let $A = 1\ 0011\ 00100000000 = (-1.125 \times 2^{-5})_{10}$

$B = 1\ 0011\ 00100000000 = (-1.125 \times 2^{-5})_{10}$,

the correct product of A multiplied by B is $(0.00123596)_{10}$.

Simulation in GFL gives the result in Figure 4.6 as:

$$W[15:0] = 0\ 0110\ 01000100000 = (0.31640625)_{10}.$$

The un = 1 indicates an exponent underflow has occurred and therefore the value in W[15 : 0] is useless. Note that no normalization is needed, the reason is x[11] = 0.

Example 6:

Let $A = 0\ 0100\ 00100000000 = (1.125 \times 2^{-4})_{10}$

$B = 0\ 0100\ 00100000000 = (1.125 \times 2^{-4})_{10}$,

the correct product of A multiplied by B is $(0.00494385)_{10}$.

Simulation in GFL gives the result in Figure 4.7 as:

$$W[15:0] = 0\ 0000\ 00000000000.$$

This result, *W[15 : 0], is forced to zero* as described in Chapter III on page 19. Note that x[11] = 0 and t_1= 1 directing MUX_0 to select zero for the result.

Example 7:

Let $A = 0\ 0000\ 00000000000$

$B = 0\ 1101\ 10101100000.$

Simulation in GFL gives the result in Figure 4.8 as:

$$W[15:0] = 0\ 0000\ 00000000000.$$

Figure 3.1 shows that t_2 = 1 directs MUX_1 to select the *true zero* result.

```
**********                                                    **********
              Genesil Screen Dump -- Tue Apr  9 23:17:25 1991

*******************************************************************************
Module: ~genluck/luck/FLPmult                                Functional Simulator
-----------------------------Genesil Version v8.0.2-----------------------------
1
b[8]
0
b[7]
0
b[6]
0
b[5]
0
b[4]
0
b[3]
0
b[2]
0
b[1]
0
b[0]
0
BACK
CYCLE
4
pi
)   is of type  module  with  22  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[15:0]  to  NC*16 = LHLHLHHHLLLLLLLL
) port 7 I b[15:0]  to  NC*16 = LHLHLHHHLLLLLLLL
) port 9 O s[15:0]  to  NC*16 = 0110110001000000
) port 11 O ov  to  NC = 0
) port 13 O cout  to  NC = 0
) port 15 O W[15:0]  to  NC*16 = 0110110001000000
) port 17 O x[11:0]  to  NC*12 = 110001000000
) port 19 O LS_OUT[11:0]  to  NC*12 = 000000000000
) port 21 O un  to  NC = 0
-------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS                  OVERLAY          RECORD      UTILITY
-------------------------------------------------------------------------------
BACK             QUERY           HIER_LEVEL      ENVIRONMENT     SCREENS
BIND             CYCLE           RUN_VECTORS     SCROLL          FIGHTS
ASSERT           STEP            UNBIND
                 PROPAGATE       VERIFY_VALUE
-------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.2 The Simulation Result of Example 1

```
**********************************************************************************
              Genesil Screen Dump -- Tue Apr  9 23:21:40 1991

**********************************************************************************
Module: ~genluck/luck/FLPmult                                  Functional Simulator
------------------------------Genesil Version v8.0.2-----------------------------
1
b[8]
0
b[7]
0
b[6]
1
b[5]
0
b[4]
0
b[3]
0
b[2]
0
b[1]
0
b[0]
0
BACK
CYCLE
4
pi
)   is of type  module  with  22  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[15:0]  to  NC*16 = LHLHLHHLLLHLLLLL
) port 7 I b[15:0]  to  NC*16 = HHLHLHHLLHLLLLLL
) port 9 O s[15:0]  to  NC*16 = 1110110010010100
) port 11 O ov  to  NC = 0
) port 13 O cout  to  NC = 0
) port 15 O W[15:0]  to  NC*16 = 1110110010010100
) port 17 O x[11:0]  to  NC*12 = 110010010100
) port 19 O LS_OUT[11:0]  to  NC*12 = 100000000000
) port 21 O un  to  NC = 0
--------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS                  OVERLAY          RECORD      UTILITY
--------------------------------------------------------------------------------
BACK              QUERY           HIER_LEVEL      ENVIRONMENT     SCREENS
BIND              CYCLE           RUN_VECTORS     SCROLL          FIGHTS
ASSERT            STEP            UNBIND
                  PROPAGATE       VERIFY_VALUE
--------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.3 The Simulation Result of Example 2

```
**********************************************************************************
              Genesil Screen Dump -- Tue Apr  9 23:38:29 1991

**********************************************************************************
Module: ~genluck/luck/FLPmult                                  Functional Simulator
------------------------------Genesil Version v8.0.2------------------------------
0
b[6]
0
b[5]
0
b[4]
0
b[3]
0
b[2]
0
b[1]
0
b[0]
0
BACK
QUERY
BACK
QUERY
BACK
CYCLE
4
pi
)   is of type module with 22 ports
) port 1 I TRUE to NC = H
) port 3 I FALSE to NC = L
) port 5 I a[15:0] to NC*16 = HHHHLHHHLLLLLLLL
) port 7 I b[15:0] to NC*16 = HHHHLHHHLLLLLLLL
) port 9 O s[15:0] to NC*16 = 0100110001000000
) port 11 O ov to NC = 1
) port 13 O cout to NC = 1
) port 15 O W[15:0] to NC*16 = 0100110001000000
) port 17 O x[11:0] to NC*12 = 110001000000
) port 19 O LS_OUT[11:0] to NC*12 = 000000000000
) port 21 O un to NC = 0
----------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS                  OVERLAY          RECORD      UTILITY
----------------------------------------------------------------------------------
BACK              QUERY           HIER_LEVEL      ENVIRONMENT     SCREENS
BIND              CYCLE           RUN_VECTORS     SCROLL          FIGHTS
ASSERT            STEP            UNBIND
                  PROPAGATE       VERIFY_VALUE
----------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.4 The Simulation Result of Example 3

```
**********************************************************************************
              Genesil Screen Dump -- Wed Apr 10 00:46:26 1991

**********************************************************************************
Module: ~genluck/luck/FLPmult                                    Functional Simulator
------------------------------Genesil Version v8.0.2------------------------------
1
b[8]
0
b[7]
0
b[6]
0
b[5]
0
b[4]
0
b[3]
0
b[2]
0
b[1]
0
b[0]
0
BACK
CYCLE
4
pi
)   is of type  module  with  22  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[15:0]  to  NC*16 = HHHLLHHLLLLLLLLL
) port 7 I b[15:0]  to  NC*16 = HHLHHHHLLLLLLLLL
) port 9 O s[15:0]  to  NC*16 = 0100010001000000
) port 11 O ov  to  NC = 1
) port 13 O cout  to  NC = 0
) port 15 O W[15:0]  to  NC*16 = 0100010001000000
) port 17 O x[11:0]  to  NC*12 = 110001000000
) port 19 O LS_OUT[11:0]  to  NC*12 = 000000000000
) port 21 O un  to  NC = 0
--------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS                  OVERLAY          RECORD      UTILITY
--------------------------------------------------------------------------------
BACK             QUERY            HIER_LEVEL       ENVIRONMENT      SCREENS
BIND             CYCLE            RUN_VECTORS      SCROLL           FIGHTS
ASSERT           STEP             UNBIND
                 PROPAGATE        VERIFY_VALUE
--------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.5 The Simulation Result of Example 4

```
**********************************************************************************
              Genesil Screen Dump -- Tue Apr  9 23:57:49 1991

**********************************************************************************
Module: ~genluck/luck/FLPmult                                  Functional Simulator
------------------------------Genesil Version v8.0.2-----------------------------
0
b[8]
1
b[7]
0
b[6]
0
b[5]
0
b[4]
0
b[3]
0
b[2]
0
b[1]
0
b[0]
0
BACK
CYCLE
4
pi
)   is of type  module  with  22  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[15:0]  to  NC*16 = HLLHHHLLHHLLLLLL
) port 7 I b[15:0]  to  NC*16 = HLLHHHLLHHLLLLLL
) port 9 O s[15:0]  to  NC*16 = 0011001000100000
) port 11 O ov  to  NC = 0
) port 13 O cout  to  NC = 0
) port 15 O W[15:0]  to  NC*16 = 0011001000100000
) port 17 O x[11:0]  to  NC*12 = 010100010000
) port 19 O LS_OUT[11:0]  to  NC*12 = 000000000000
) port 21 O un  to  NC = 1
----------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS                  OVERLAY          RECORD       UTILITY
----------------------------------------------------------------------------------
BACK             QUERY            HIER_LEVEL      ENVIRONMENT      SCREENS
BIND             CYCLE            RUN_VECTORS     SCROLL           FIGHTS
ASSERT           STEP             UNBIND
                 PROPAGATE        VERIFY_VALUE
----------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.6 The Simulation Result of Example 5

```
**********************************************************************************
              Genesil Screen Dump -- Tue Apr  9 23:54:05 1991

**********************************************************************************
Module: ~genluck/luck/FLPmult                                    Functional Simulator
------------------------------Genesil Version v8.0.2-----------------------------
1
b[9]
0
b[7]
0
b[6]
0
b[5]
0
b[4]
0
b[3]
0
b[2]
0
b[1]
0
b[0]
0
BACK
CYCLE
4
pi
)   is of type  module  with  22  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[15:0]  to  NC*16 = LLHLLLLHLLLLLLL
) port 7 I b[15:0]  to  NC*16 = LLHLLLLHLLLLLLL
) port 9 O s[15:0]  to  NC*16 = 0000000000000000
) port 11 O ov  to  NC = 0
) port 13 O cout  to  NC = 1
) port 15 O W[15:0]  to  NC*16 = 0000000000000000
) port 17 O x[11:0]  to  NC*12 = 010100010000
) port 19 O LS_OUT[11:0]  to  NC*12 = 000000000000
) port 21 O un  to  NC = 0
---------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS                  OVERLAY          RECORD       UTILITY
---------------------------------------------------------------------------------
BACK              QUERY            HIER_LEVEL      ENVIRONMENT      SCREENS
BIND              CYCLE            RUN_VECTORS     SCROLL           FIGHTS
ASSERT            STEP             UNBIND
                  PROPAGATE        VERIFY_VALUE
---------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.7 The Simulation Result of Example 6

```
**********************************************************************************
              Genesil Screen Dump -- Wed Apr 10 00:04:12 1991

**********************************************************************************
Module: ~genluck/luck/FLPmult                                  Functional Simulator
------------------------------Genesil Version v8.0.2------------------------------
0
b[8]
1
b[7]
0
b[6]
1
b[5]
1
b[4]
0
b[3]
0
b[2]
0
b[1]
0
b[0]
0
BACK
CYCLE
4
pi
)   is of type  module  with  22  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[15:0]  to  NC*16 = LLLLLLLLLLLLLLLL
) port 7 I b[15:0]  to  NC*16 = LHHLHHHLHHLHHLLL
) port 9 O s[15:0]  to  NC*16 = 0010110101100000
) port 11 O ov  to  NC = 0
) port 13 O cout  to  NC = 0
) port 15 O W[15:0]  to  NC*16 = 0000000000000000
) port 17 O x[11:0]  to  NC*12 = 011010110000
) port 19 O LS_OUT[11:0]  to  NC*12 = 000000000000
) port 21 O un  to  NC = 0
--------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS                  OVERLAY          RECORD      UTILITY
--------------------------------------------------------------------------------
BACK              QUERY           HIER_LEVEL      ENVIRONMENT     SCREENS
BIND              CYCLE           RUN_VECTORS     SCROLL          FIGHTS
ASSERT            STEP            UNBIND
                  PROPAGATE       VERIFY_VALUE
--------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.8 The Simulation Result of Example 7

B. THE SIMULATION RESULT OF FLOATING-POINT ADD UNIT

Example 8:

Let A and B be the two input operands of the floating-point ADD unit. Assume

$$A = 0\ 1010\ 11000000000 = (1.75 \times 2^2)_{10}$$

$$\text{and } B = 0\ 1010\ 11000000000 = (1.75 \times 2^2)_{10}.$$

When simulation is executed on the addition of 7.0 plus 7.0, the sum of A and B is shown in the Figure 4.9

$$D_{out}\,[15:0] = 0\ 1011\ 11000000000$$

that can be verified to be the value of $1.75 \times 2^3 = 14$.

Note that Ae[3 : 0] = 0000 illustrates the exponent parts of two input operands are equal. Thus no alignment is needed. Both ov and un are zero showing that neither an overflow or an underflow has occurred. The result does not suffer from truncation error in this example.

Example 9:

Let $A = 0\ 1001\ 11000000010 = (1.75097656 \times 2)_{10}$

$B = 0\ 1100\ 11000000000 = (1.75 \times 2^4)_{10}$,

the correct sum of A plus B is $(31.501953125)_{10}$.

Simulation in GFL gives the result in Figure 4.10 as:

$$D_{out}\,[15:0] = 0\ 1100\ 11111000000 = (31.5)_{10}.$$

The Ae[3] = 1 illustrates A<B, and A will be aligned. Note that the difference value, 0.001953125 (= 31.501953125 – 31.5) is due to truncation error that is caused by alignment.

Example 10:

Let $A = 0\ 1000\ 01111110110 = (1.4951171875)_{10}$

$B = 1\ 0100\ 10000100000 = (-1.515625 \times 2^{-4})_{10}$,

the correct sum of A plus B is $(1.400390625)_{10}$.

Simulation in GFL gives the result in Figure 4.11 as:

$D_{out}[15:0] = 0\ 1000\ 01100110100 = (1.400390625)_{10}$.

This example shows a true subtraction, (+A) + (–B), as described in Chapter III on page 24. The sel = 1 converts the sum of fraction part (i.e., K[12 : 0]) to sign-magnitude form (i.e., m[12 : 0]).

Example 11:

Let $A = 1\ 0110\ 01110000100 = (-1.439453125 \times 2^{-2})_{10}$

$B = 0\ 0110\ 10001000000 = (1.53125 \times 2^{-2})_{10}$,

the correct sum of A plus B is $(0.02294921875)_{10}$.

Simulation in GFL gives the result in Figure 4.12 as:

$D_{out}[15:0] = 0\ 0010\ 01111000000 = (1.53125 \times 2^{-2})_{10}$.

The g[3 : 0] = 0100 illustrates that normalization has occurred, and shifting four digits left is executed in the final fraction part (i.e., f[10 : 0]). Note that the exponent part (i.e., de[3 : 0]) has adjusted as part of the normalization process.

Example 12:

Let $A = 0\ 0001\ 11101000000 = (1.90625 \times 2^{-7})_{10}$

$B = 1\ 0001\ 00111111000 = (-1.18359375 \times 2^{-7})_{10}$,

the correct sum of A plus B is $(0.00564575195)_{10}$.

Simulation in GFL gives the result in Figure 4.13 as:

$D_{out}\ [15:0] = 0\ 1111\ 10100100000 = (210)_{10}$.

The un = 1 indicates an exponent underflow has occurred and therefore the value in $D_{out}[15:0]$ is useless.

Example 13:

Let $A = 1\ 1111\ 11000000000 = (-1.75 \times 2^{7})_{10}$

$B = 1\ 1111\ 10010000000 = (-1.5625 \times 2^{7})_{10}$,

the correct sum of A plus B is $(-424)_{10}$.

Simulation in GFL gives the result in Figure 4.14 as:

$D_{out}\ [15:0] = 1\ 0000\ 00000000000$.

The ov = 1 indicates an exponent overflow has occurred and therefore the value in $D_{out}[15:0]$ is useless.

Example 14:

Let $A = 0\ 0000\ 00000000000 = (0)_{10}$

$B = 1\ 1100\ 01000100000 = (-1.265625 \times 2^{4})_{10}$,

the correct sum of A plus B is $(-20.25)_{10}$.

Simulation in GFL gives the result in Figure 4.15 as:

$D_{out}\ [15:0] = 1\ 1100\ 01000100000 = (-20.25)_{10}$.

Note that the *true zero* operand, A, which is successfully detected.

Example 15:

Let $A = 0\ 1010\ 11000000000 = (1.75 \times 2^4)_{10}$

$B = 1\ 1010\ 11000000000 = (-1.75 \times 2^4)_{10,}$

the correct sum of A plus B is $(0)_{10.}$

Simulation in GFL gives the result in Figure 4.16 as:

$$D_{out}\,[15:0] = 0\ 0000\ 00000000000 = (0)_{10.}$$

Note that this example shows (A) + (–A) = 0.

```
**********************************************************************************
              Genesil Screen Dump -- Thu Apr 11 10:58:46 1991

**********************************************************************************
Module: ~genluck/luck/FLPadder                                  Functional Simulator
-----------------------------Genesil Version v8.0.2-----------------------------
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 O Cout  to  NC = 0
) port 7 O Dout[15:0]  to  NC*16 = 0101111000000000
) port 9 O Ae[3:0]  to  NC*4 = 0000
) port 11 O Be[3:0]  to  NC*4 = 0000
) port 13 O ae[3:0]  to  NC*4 = 1010
) port 15 CI phase_a  to  NC = 1
) port 17 CI phase_b  to  NC = 0
) port 19 O t2  to  NC = 0
) port 21 O be[3:0]  to  NC*4 = 1010
) port 23 O SE[3:0]  to  NC*4 = 0000
) port 25 O align[11:0]  to  NC*12 = 111000000000
) port 27 O am[10:0]  to  NC*11 = 11000000000
) port 29 O bm[10:0]  to  NC*11 = 11000000000
) port 31 O ce[3:0]  to  NC*4 = 1010
) port 33 O K[12:0]  to  NC*13 = 1110000000000
) port 35 O Bb[12:0]  to  NC*13 = 0111000000000
) port 37 I A[15:0]  to  NC*16 = LHLHLHHLLLLLLLLL
) port 39 I B[15:0]  to  NC*16 = LHLHLHHLLLLLLLLL
) port 41 O abs  to  NC = 0
) port 43 O aligned[11:0]  to  NC*12 = 111000000000
) port 45 O as  to  NC = 0
) port 47 O bs  to  NC = 0
) port 49 O sel  to  NC = 0
) port 51 O sign  to  NC = 0
) port 53 O f[10:0]  to  NC*11 = 11000000000
) port 55 O de[3:0]  to  NC*4 = 1011
) port 57 O t1  to  NC = 0
) port 59 O g[3:0]  to  NC*4 = 0000
) port 61 O m[12:0]  to  NC*13 = 1110000000000
) port 63 O d[3:0]  to  NC*4 = 0001
) port 65 O e[3:0]  to  NC*4 = 1011
) port 67 O ov  to  NC = 0
) port 69 O un  to  NC = 0
--------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS                OVERLAY          RECORD       UTILITY
--------------------------------------------------------------------------------
BACK              QUERY           HIER_LEVEL      ENVIRONMENT     SCREENS
BIND              CYCLE           RUN_VECTORS     SCROLL          FIGHTS
ASSERT            STEP            UNBIND
                  PROPAGATE       VERIFY_VALUE
--------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.9 The Simulation Result of Example 8

```
**********************************************************************************
              Genesil Screen Dump -- Thu Apr 11 11:17:10 1991
**********************************************************************************
Module: ~genluck/luck/FLPadder                                  Functional Simulator
-----------------------------Genesil Version v8.0.2-----------------------------
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 O Cout  to  NC = 0
) port 7 O Dout[15:0]  to  NC*16 = 0110011111000000
) port 9 O Ae[3:0]  to  NC*4 = 1101
) port 11 O Be[3:0]  to  NC*4 = 0011
) port 13 O ae[3:0]  to  NC*4 = 1001
) port 15 CI phase_a  to  NC = 1
) port 17 CI phase_b  to  NC = 0
) port 19 O t2  to  NC = 0
) port 21 O be[3:0]  to  NC*4 = 1100
) port 23 O SE[3:0]  to  NC*4 = 0011
) port 25 O align[11:0]  to  NC*12 = 111000000010
) port 27 O am[10:0]  to  NC*11 = 11000000010
) port 29 O bm[10:0]  to  NC*11 = 11000000000
) port 31 O ce[3:0]  to  NC*4 = 1100
) port 33 O K[12:0]  to  NC*13 = 0111111000000
) port 35 O Bb[12:0]  to  NC*13 = 0111000000000
) port 37 I A[15:0]  to  NC*16 = LHLLHHHLLLLLLLHL
) port 39 I B[15:0]  to  NC*16 = LHHLLHHLLLLLLLLL
) port 41 O abs  to  NC = 0
) port 43 O aligned[11:0]  to  NC*12 = 000111000000
) port 45 O as  to  NC = 0
) port 47 O bs  to  NC = 0
) port 49 O sel  to  NC = 0
) port 51 O sign  to  NC = 0
) port 53 O f[10:0]  to  NC*11 = 11111000000
) port 55 O de[3:0]  to  NC*4 = 1100
) port 57 O t1  to  NC = 0
) port 59 O g[3:0]  to  NC*4 = 0000
) port 61 O m[12:0]  to  NC*13 = 0111111000000
) port 63 O d[3:0]  to  NC*4 = 0000
) port 65 O e[3:0]  to  NC*4 = 1100
) port 67 O ov  to  NC = 0
) port 69 O un  to  NC = 0
----------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS          OVERLAY          RECORD       UTILITY
----------------------------------------------------------------------------------
BACK              QUERY             HIER_LEVEL       ENVIRONMENT  SCREENS
BIND              CYCLE             RUN_VECTORS      SCROLL       FIGHTS
ASSERT            STEP              UNBIND
                  PROPAGATE         VERIFY_VALUE
----------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.10 The Simulation Result of Example 9

```
**********************************************************************************
              Genesil Screen Dump -- Thu Apr 11 11:22:04 1991

**********************************************************************************
Module: ~genluck/luck/FLPadder                                  Functional Simulator
------------------------------Genesil Version v8.0.2-----------------------------
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 O Cout  to  NC = 0
) port 7 O Dout[15:0]  to  NC*16 = 0100001100110100
) port 9 O Ae[3:0]  to  NC*4 = 0100
) port 11 O Be[3:0]  to  NC*4 = 1100
) port 13 O ae[3:0]  to  NC*4 = 1000
) port 15 CI phase_a  to  NC = 1
) port 17 CI phase_b  to  NC = 0
) port 19 O t2  to  NC = 0
) port 21 O be[3:0]  to  NC*4 = 0100
) port 23 O SE[3:0]  to  NC*4 = 0100
) port 25 O align[11:0]  to  NC*12 = 110000100000
) port 27 O am[10:0]  to  NC*11 = 01111110110
) port 29 O bm[10:0]  to  NC*11 = 10000100000
) port 31 O ce[3:0]  to  NC*4 = 1000
) port 33 O K[12:0]  to  NC*13 = 1010011001100
) port 35 O Bb[12:0]  to  NC*13 = 0101111110110
) port 37 I A[15:0]  to  NC*16 = LHLLLLLHHHHHHHLHHL
) port 39 I B[15:0]  to  NC*16 = HLHLLHLLLLJHLLLLL
) port 41 O abs  to  NC = 1
) port 43 O aligned[11:0]  to  NC*12 = 000011000010
) port 45 O as  to  NC = 0
) port 47 O bs  to  NC = 1
) port 49 O sel  to  NC = 1
) port 51 O sign  to  NC = 0
) port 53 O f[10:0]  to  NC*11 = 01100110100
) port 55 O de[3:0]  to  NC*4 = 1000
) port 57 O t1  to  NC = 0
) port 59 O g[3:0]  to  NC*4 = 0000
) port 61 O m[12:0]  to  NC*13 = 0101100110100
) port 63 O d[3:0]  to  NC*4 = 0000
) port 65 O e[3:0]  to  NC*4 = 1000
) port 67 O ov  to  NC = 0
) port 69 O un  to  NC = 0
----------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS                  OVERLAY          RECORD       UTILITY
----------------------------------------------------------------------------------
BACK              QUERY            HIER_LEVEL      ENVIRONMENT     SCREENS
BIND              CYCLE            RUN_VECTORS     SCROLL          FIGHTS
ASSERT            STEP             UNBIND
                  PROPAGATE        VERIFY_VALUE
----------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.11 The Simulation Result of Example 10

```
**********************************************************************************
              Genesil Screen Dump -- Thu Apr 11 11:29:38 1991

**********************************************************************************
Module: ~genluck/luck/FLPadder                                  Functional Simulator
------------------------------Genesil Version v8.0.2-------------------------------
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 O Cout  to  NC = 1
) port 7 O Dout[15:0]  to  NC*16 = 0001001111000000
) port 9 O Ae[3:0]  to  NC*4 = 0000
) port 11 O Be[3:0]  to  NC*4 = 0000
) port 13 O ae[3:0]  to  NC*4 = 0110
) port 15 CI phase_a  to  NC = 1
) port 17 CI phase_b  to  NC = 0
) port 19 O t2  to  NC = 0
) port 21 O be[3:0]  to  NC*4 = 0110
) port 23 O SE[3:0]  to  NC*4 = 0000
) port 25 O align[11:0]  to  NC*12 = 110001000000
) port 27 O am[10:0]  to  NC*11 = 01110000100
) port 29 O bm[10:0]  to  NC*11 = 10001000000
) port 31 O ce[3:0]  to  NC*4 = 0110
) port 33 O K[12:0]  to  NC*13 = 1111101000100
) port 35 O Bb[12:0]  to  NC*13 = 1101110000100
) port 37 I A[15:0]  to  NC*16 = HLHHLLHHHHLLLLHLL
) port 39 I B[15:0]  to  NC*16 = LLHHLHLLLLHLLLLL
) port 41 O abs  to  NC = 0
) port 43 O aligned[11:0]  to  NC*12 = 110001000000
) port 45 O as  to  NC = 1
) port 47 O bs  to  NC = 0
) port 49 O sel  to  NC = 1
) port 51 O sign  to  NC = 0
) port 53 O f[10:0]  to  NC*11 = 01111000000
) port 55 O de[3:0]  to  NC*4 = 0010
) port 57 O t1  to  NC = 0
) port 59 O g[3:0]  to  NC*4 = 0100
) port 61 O m[12:0]  to  NC*13 = 0000010111100
) port 63 O d[3:0]  to  NC*4 = 1100
) port 65 O e[3:0]  to  NC*4 = 0010
) port 67 O ov  to  NC = 0
) port 69 O un  to  NC = 0
----------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS                  OVERLAY          RECORD       UTILITY
----------------------------------------------------------------------------------
BACK              QUERY            HIER_LEVEL       ENVIRONMENT      SCREENS
BIND              CYCLE            RUN_VECTORS      SCROLL           FIGHTS
ASSERT            STEP             UNBIND
                  PROPAGATE        VERIFY_VALUE
----------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.12 The Simulation Result of Example 11

```
**********************************************************************************
              Genesil Screen Dump -- Thu Apr 11 11:59:42 1991

**********************************************************************************
Module: ~genluck/luck/FLPadder                                    Functional Simulator
-----------------------------Genesil Version v8.0.2-----------------------------
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 O Cout  to  NC = 0
) port 7 O Dout[15:0]  to  NC*16 = 0111110100100000
) port 9 O Ae[3:0]  to  NC*4 = 0000
) port 11 O Be[3:0]  to  NC*4 = 0000
) port 13 O ae[3:0]  to  NC*4 = 0001
) port 15 CI phase_a  to  NC = 1
) port 17 CI phase_b  to  NC = 0
) port 19 O t2  to  NC = 0
) port 21 O be[3:0]  to  NC*4 = 0001
) port 23 O SE[3:0]  to  NC*4 = 0000
) port 25 O align[11:0]  to  NC*12 = 100111111000
) port 27 O am[10:0]  to  NC*11 = 11101000000
) port 29 O bm[10:0]  to  NC*11 = 00111111000
) port 31 O ce[3:0]  to  NC*4 = 0001
) port 33 O K[12:0]  to  NC*13 = 1101010111000
) port 35 O Bb[12:0]  to  NC*13 = 0111101000000
) port 37 I A[15:0]  to  NC*16 = LLLLHHHHHLHLLLLL
) port 39 I B[15:0]  to  NC*16 = HLLLHLLHHHHHHHLL
) port 41 O abs  to  NC = 1
) port 43 O aligned[11:0]  to  NC*12 = 100111111000
) port 45 O as  to  NC = 0
) port 47 O bs  to  NC = 1
) port 49 O sel  to  NC = 1
) port 51 O sign  to  NC = 0
) port 53 O f[10:0]  to  NC*11 = 10100100000
) port 55 O de[3:0]  to  NC*4 = 1111
) port 57 O t1  to  NC = 0
) port 59 O g[3:0]  to  NC*4 = 0010
) port 61 O m[12:0]  to  NC*13 = 0010101001000
) port 63 O d[3:0]  to  NC*4 = 1110
) port 65 O e[3:0]  to  NC*4 = 1111
) port 67 O ov  to  NC = 0
) port 69 O un  to  NC = 1
----------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS                  OVERLAY          RECORD       UTILITY
----------------------------------------------------------------------------------
BACK              QUERY           HIER_LEVEL       ENVIRONMENT      SCREENS
BIND              CYCLE           RUN_VECTORS      SCROLL           FIGHTS
ASSERT            STEP            UNBIND
                  PROPAGATE       VERIFY_VALUE
----------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.13 The Simulation Result of Example 12

```
**********
              Genesil Screen Dump -- Thu Apr 11 12:03:18 1991

**********
Module: ~genluck/luck/FLPadder                                  Functional Simulator
-----------------------------Genesil Version v8.0.2-----------------------------
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 O Cout  to  NC = 1
) port 7 O Dout[15:0]  to  NC*16 = 1000000000000000
) port 9 O Ae[3:0]  to  NC*4 = 0000
) port 11 O Be[3:0]  to  NC*4 = 0000
) port 13 O ae[3:0]  to  NC*4 = 1111
) port 15 CI phase_a  to  NC = 1
) port 17 CI phase_b  to  NC = 0
) port 19 O t2  to  NC = 0
) port 21 O be[3:0]  to  NC*4 = 1111
) port 23 O SE[3:0]  to  NC*4 = 0000
) port 25 O align[11:0]  to  NC*12 = 110010000000
) port 27 O am[10:0]  to  NC*11 = 11000000000
) port 29 O bm[10:0]  to  NC*11 = 10010000000
) port 31 O ce[3:0]  to  NC*4 = 1111
) port 33 O K[12:0]  to  NC*13 = 1101010000000
) port 35 O Bb[12:0]  to  NC*13 = 1111000000000
) port 37 I A[15:0]  to  NC*16 = HHHHHHHHHLLLLLLLL
) port 39 I B[15:0]  to  NC*16 = HHHHHHHHLLHHLLLLLL
) port 41 O abs  to  NC = 1
) port 43 O aligned[11:0]  to  NC*12 = 110010000000
) port 45 O as  to  NC = 1
) port 47 O bs  to  NC = 1
) port 49 O sel  to  NC = 0
) port 51 O sign  to  NC = 1
) port 53 O f[10:0]  to  NC*11 = 00000000000
) port 55 O de[3:0]  to  NC*4 = 0000
) port 57 O t1  to  NC = 1
) port 59 O g[3:0]  to  NC*4 = 0000
) port 61 O m[12:0]  to  NC*13 = 1101010000000
) port 63 O d[3:0]  to  NC*4 = 0001
) port 65 O e[3:0]  to  NC*4 = 0000
) port 67 O ov  to  NC = 1
) port 69 O un  to  NC = 0
------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS                  OVERLAY          RECORD      UTILITY
------------------------------------------------------------------------------
BACK              QUERY            HIER_LEVEL       ENVIRONMENT      SCREENS
BIND              CYCLE            RUN_VECTORS      SCROLL           FIGHTS
ASSERT            STEP             UNBIND
                  PROPAGATE        VERIFY_VALUE
------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.14 The Simulation Result of Example 13

```
**********************************************************************************
              Genesil Screen Dump -- Thu Apr 11 12:07:28 1991

**********************************************************************************
Module: ~genluck/luck/FLPadder                                  Functional Simulator
------------------------------Genesil Version v8.0.2------------------------------
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 O Cout  to  NC = 0
) port 7 O Dout[15:0]  to  NC*16 = 1110001000100000
) port 9 O Ae[3:0]  to  NC*4 = 0100
) port 11 O Be[3:0]  to  NC*4 = 1100
) port 13 O ae[3:0]  to  NC*4 = 0000
) port 15 CI phase_a  to  NC = 1
) port 17 CI phase_b  to  NC = 0
) port 19 O t2  to  NC = 1
) port 21 O be[3:0]  to  NC*4 = 1100
) port 23 O SE[3:0]  to  NC*4 = 0100
) port 25 O align[11:0]  to  NC*12 = 101000100000
) port 27 O am[10:0]  to  NC*11 = 00000000000
) port 29 O bm[10:0]  to  NC*11 = 01000100000
) port 31 O ce[3:0]  to  NC*4 = 0000
) port 33 O K[12:0]  to  NC*13 = 1100010100010
) port 35 O Bb[12:0]  to  NC*13 = 0100000000000
) port 37 I A[15:0]  to  NC*16 = LLLLLLLLLLLLLLLL
) port 39 I B[15:0]  to  NC*16 = HHHHLLLHLLLHLLLL
) port 41 O abs  to  NC = 1
) port 43 O aligned[11:0]  to  NC*12 = 000010100010
) port 45 O as  to  NC = 0
) port 47 O bs  to  NC = 1
) port 49 O sel  to  NC = 1
) port 51 O sign  to  NC = 0
) port 53 O f[10:0]  to  NC*11 = 10101111000
) port 55 O de[3:0]  to  NC*4 = 1110
) port 57 O t1  to  NC = 0
) port 59 O g[3:0]  to  NC*4 = 0010
) port 61 O m[12:0]  to  NC*13 = 0011101011110
) port 63 O d[3:0]  to  NC*4 = 1110
) port 65 O e[3:0]  to  NC*4 = 1110
) port 67 O ov  to  NC = 0
) port 69 O un  to  NC = 1
--------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS                OVERLAY          RECORD       UTILITY
--------------------------------------------------------------------------------
BACK              QUERY           HIER_LEVEL      ENVIRONMENT     SCREENS
BIND              CYCLE           RUN_VECTORS     SCROLL          FIGHTS
ASSERT            STEP            UNBIND
                  PROPAGATE       VERIFY_VALUE
--------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.15 The Simulation Result of Example 14

```
*******************************************************************************
              Genesil Screen Dump -- Mon May  6 16:50:35 1991

*******************************************************************************
Module: ~genluck/luck/FLPadder                                Functional Simulator
------------------------------Genesil Version v8.0.2--------------------------------
0
B[10]
1
B[9]
1
B[8:0]
000000000
BACK
CYCLE
4
pi
)   is of type  module  with  182  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 O Cout  to  NC = 0
) port 7 O Dout[15:0]  to  NC*16 = 0000000000000000
) port 9 O Ae[3:0]  to  NC*4 = 0000
) port 11 O Be[3:0]  to  NC*4 = 0000
) port 13 O ae[3:0]  to  NC*4 = 1010
) port 15 CI phase_a  to  NC = 1
) port 17 CI phase_b  to  NC = 0
) port 19 O t2  to  NC = 0
) port 21 O be[3:0]  to  NC*4 = 1010
) port 23 O SE[3:0]  to  NC*4 = 0000
) port 25 O align[11:0]  to  NC*12 = 111000000000
) port 27 O am[10:0]  to  NC*11 = 11000000000
) port 29 O bm[10:0]  to  NC*11 = 11000000000
) port 31 O ce[3:0]  to  NC*4 = 1010
) port 33 O Q[15:0]  to  NC*16 = 0101011000000000
) port 35 O K[12:0]  to  NC*13 = 0000000000000
) port 37 O Bb[12:0]  to  NC*13 = 0111000000000
) port 39 O n[11:0]  to  NC*12 = 000000000000
) port 41 I A[15:0]  to  NC*16 = LHLHLHHLLLLLLLLL
) port 43 I B[15:0]  to  NC*16 = HHLHLHHLLLLLLLLL
) port 45 O abs  to  NC = 1
-------------------------------------------------------------------------------
 INSERT  MESSAGES  GRAPHICS                  OVERLAY          RECORD       UTILITY
-------------------------------------------------------------------------------
BACK             QUERY          HIER_LEVEL      ENVIRONMENT     SCREENS
BIND             CYCLE          RUN_VECTORS     SCROLL          FIGHTS
ASSERT           STEP           UNBIND
                 PROPAGATE      VERIFY_VALUE
-------------------------------------------------------------------------------

Command:
>SIMULATION>
```

Figure 4.16 The Simulation Result of Example 15

V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

The purpose of this thesis was to design the floating-point hardware of the multiplier, adder and subtracter for FFT operation. The Genesil Silicon Compiler system was used to overcome the shortcomings of the time consuming custom graphical layout methods. The Genesil Silicon Compiler system (V 8.0.2), currently licensed at the Naval Postgraduate School, provides IC designers with the capability to extend the Genesil Silicon Compiler Library with fully parameterized cells that work with Genesil verification and floorplanning tools [Ref. 13]. The Genesil Silicon Compiler system greatly increases the ability to verify the designs and aids in wiring.

A 16 bit reduced word size floating-point arithmetic unit for high speed signal analysis was implemented in this thesis. The design algorithm for floating-point arithmetic units is introduced in Chapter II; the algorithm also supports the hardware design of floating-point arithmetic units that are described in Chapter III. Other efforts of this thesis are the layout verification, functional simulation result, and Timing Analysis (TA) [Ref. 16] of the floating-point units. The validated designs can be used to develop the high speed, pipelined floating-point FFT units.

Table 5.1 shows the timing analysis of the floating-point MULTIPLY unit and ADD unit. Note that in Table 5.1 the worst case delay of the floating-point MULTIPLY unit is 64.4 ns, while the delay of the ADD unit is 215.8 ns. That is, the maximum clock rates are about 15.5 MHz for multiplier and 4.6

MHz for adder. Table 5.1 also shows the floating-point ADD unit has greater area than the floating-point MULTIPLY unit.

TABLE 5.1 THE OUTPUT DELAY AND SIZE FOR FLOATING-POINT MULTIPLY AND ADD

Floating-Point Unit	Output Delay		Size(Mils)		Fabline
	min(ns)	max(ns)	height	width	
MULTIPLY	3.4	64.4	101.4	101.5	VTC_10PE
ADD	21.7	215.8	122.3	430.8	VTC_10PE

B. RECOMMENDATIONS

Notice that the maximum output delay of the adder illustrates the operating frequency is not high enough. To speed up the adder and to balance clock rates is a task to be investigated. The goal of the designer is to make the floating-point adder as fast as the multiplier. Two possible improvements are

- Rearrange or redesign the hardware of floating-point ADD unit, and
- Pipeline the floating-point ADD unit.

Note that redesign of the hardware of floating-point ADD unit could be a difficult task. This requires a through study of the delay bottleneck. However, based on our experience, the marginal improvement is slight and the natural approach is to pipeline the adder at the expense of silicon area. When pipelined, the ADD unit would have approximately four times the

number of stages as the MULTIPLY unit. It is recommended that pipelined designs be developed following the basic work in this thesis.

In this thesis, if the value is $e = 2^{-8}$ and $f \neq 0$ (see Figure 2.1 on page 7) then this value of the result will be forced to *zero*. If the value is $e < 2^{-8}$ then we only set the underflow flag. The designer can redesign the circuit so that when the exponent underflow occurs, the output value will be forced to *true zero*. Now, a *true zero* will be forced as output when the actual exponent would have been less than or equal to 2^{-8}. The underflow flag should be set in both cases.

Special purpose, reduced word size, floating-point multipliers and adders, are useful units for high speed signal processing, particularly in FFT units. Therefore the work should be continued.

LIST OF REFERENCES

1. Brigham, E. O., *The fast Fourier transfer and its applications,* pp. 1-8, PrenticHall, 1988.

2. *Genesil System, System Description Users Manual,* Silicon Complier Systems Corp., San Jose, CA, 1986.

3. Huber, R. S., *Design of A Pipelined Multiplier Using A Silicon Compiler,* Master's thesis, Naval Postgraduate School, Monterey, CA, June 1990.

4. Kung, C. F., *A Pipelined Implementation of Notch Filters Using Genesil Silicon Complier,* Master's thesis, Naval Postgraduate School, Monterey, CA, March 1990.

5. John, L. H., and David, A. P., *Computer Architecture A Quantitative Approach,* pp. A12-A28, Morgan Kaufmann Publishing Inc., 1990.

6. Pollard, L. H., *Computer design and archutecture,* pp. 69-122, Prentice Hall, 1990.

7. Strum, R. D., and Kirk, D. E., *Discrete Systems and Digital Signal Processing,* pp. 495-516, Addision-Wesely Inc., April 1989.

8. Fredrick, J. H and Gerald, R. P., *Digital System Hardware Organization and Design,* pp. 557-569, John Wiley & Sons, Inc., 1987.

9. Hwang, K., *Computer Arithmetic Principles, Architecture, and Design,* pp. 285-308, John Wiley & Sons, Inc., 1979.

10. Weste, N., *Principles of CMOS VLSI Design,* Addision-Wesely Inc., pp. 236-259, June 1988.

11. Cody, W. J., *A Proposed Standard for Binary Floating-Point Arithmetic,* pp. 51-66, IEEE comput. Soc. Press, March 1981.

12. Stuart, D. C., *VLSI Designs for Piplined FFT Processors,* Master's thesis, Naval Postgraduate School, Monterey, CA, June 1990.

13. Settle, R. H., *Design Methodology Using The Genesil Silicon Compiler, Master's thesis,* Naval Postgraduate School, Monterey, CA, September 1988.

14. *Genesil System, System Description Application Commands,* Silicon Complier Systems Corp., San Jose, CA, 1987.

15. *Genesil System, Simulation Users Guide,* Silicon Complier Systems Corp., San Jose, CA, 1986.

16. *Genesil System, Timing Analysis User Guide,* Silicon Complier Systems Corp., San Jose, CA, 1987.

17. *Genesil System Compiler Library Volume I Blocks,* Silicon Complier Systems Corp., San Jose, CA, 1986.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2. Library, Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
3. Department Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
4. Prof. Chyan Yang, Code EC/Ya Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	3
5. Prof. Herschel H. Loomis, Jr., Code EC/Lm Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	3
6. Lu, Chung-Kuei 347 Her-Chyan Village Tsoying Kaoshiung Taiwan, Republic of China	1
7. T.F.P.G Library P.O. Box 8761 Ta-Fu, I-Lan Taiwan, Republic of China	1
8. Library of Chinese Naval Academy P.O. Box 8494 Tsoying Kaohsiung, Taiwan, Republic of China	2